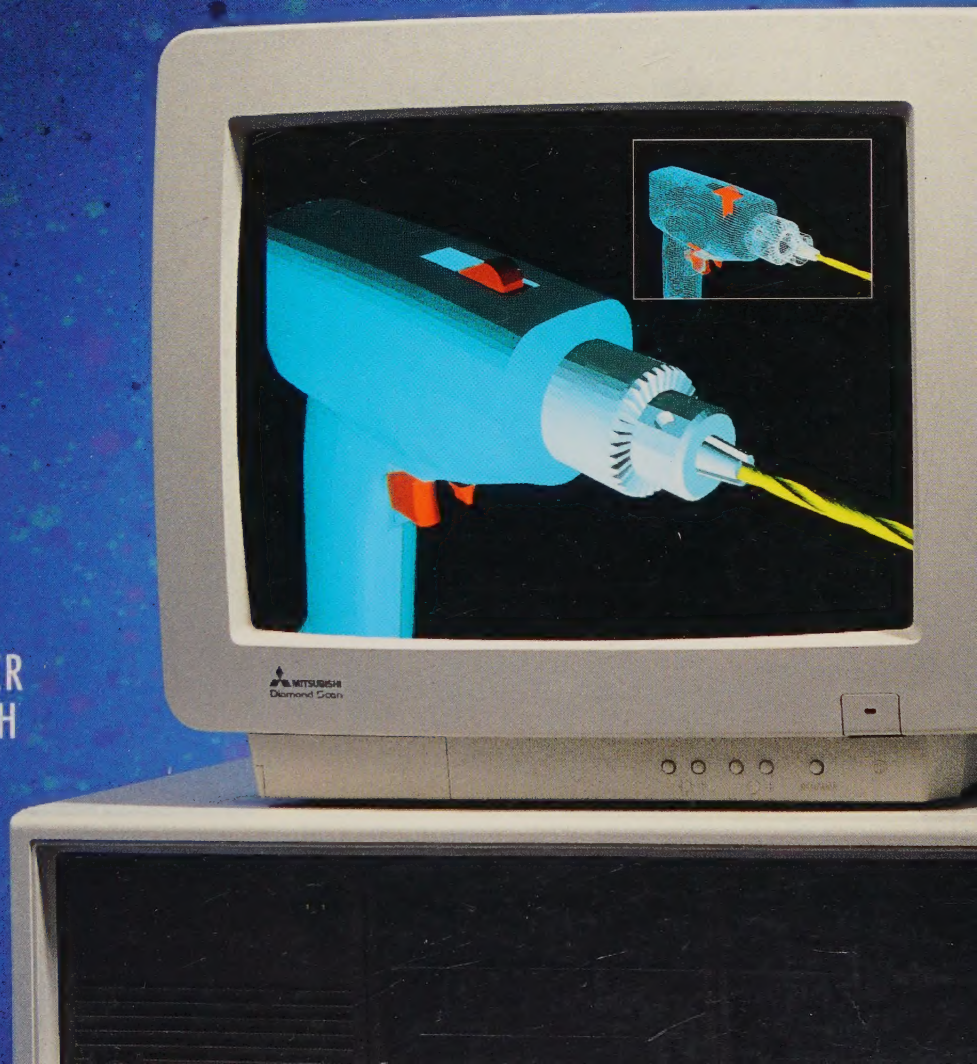
GRAPHICS PROGRAMMING FOR THE 8514/A

THE NEW PC GRAPHICS STANDARD



JAKE RICHTER
& BUD SMITH

Digitized by the Internet Archive
in 2024

https://archive.org/details/graphicsprogramm0000rich

Graphics Programming for the 8514/A

M&T Publishing, Inc.

Graphics Programming for the 8514/A

Jake Richter
and
Bud Smith

M&T Publishing, Inc.
Redwood City, California

M&T Books
A Division of M&T Publishing, Inc.
501 Galveston Drive
Redwood City, CA 94063

© 1990 by M&T Publishing, Inc.

Printed in the United States of America
First Edition published 1989

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without prior written permission from the Publisher. Contact the Publisher for information on foreign rights.

ISBN 1-55851-079-6 (book/disk)
ISBN 1-55851-074-5 (book)
ISBN 1-55851-104-0 (disk)

93 92 91 90 4 3 2 1

Trademarks
Many of the products described in this book are only now coming into production, and their names have not yet been registered as trademarks at this writing. We recognize all trademarks, registered or unregistered, claimed by the manufacturers of the products discussed in this book. The use of these names in the book is not intended as infringement on any trademark.

Project Editor: David Rosenthal

To our wives

for their love and support through the many trips and long hours
required to research and write this book, and for threatening
to take away our toys if we didn't get the book
completed in a timely fashion.

Contents

Why this Book Is for You **1**

Introduction **3**
Who You Are 5
Who We Are 6
How the Book Is Organized 8
Tools and Resources 9
Keeping Updated 9

1. PC Graphics Standards **11**
The Need for a Better Standard 14
The AI vs. Hardware Compatibility 16
How the 8514/A Fits In 17
Market Prospects for the 8514/A 18
The TI 34010 and 34020 20
Benchmarking 8514/A and 34010 Boards 23
Memory Math 25
Monitor Requirements 28
Monitor Math 30
The PC Graphics Industry 31
The Role of VESA 32

2. IBM's 8514/A **35**
Features of the 8514/A 38
Architecture of the IBM 8514/A 40
Color math 42
Implementing the Architecture 42
The Processor and Registers 43
Mixes 47

Memory Architecture 48
Font Support 49
The 8514/A and VGA 51
Functional Highlights 53
Functional Limitations 55

3. Software Development for the 8514/A 57
Standards 57
Performance 58
Text Display 60
Compatibles and VGA Support 61
Portability 62
Ease of Development and Maintenance 63
A Development Strategy 64

4. Programming the AI 67
Accessing the AI 68
Function Parameter Block 69
Coordinates 71
Combining the AI and the Registers 72
AI Speed 72
The AI Commands 73

5. Tutorial: The AI 157
What's First? 158
Opening the Graphics Device 159
Closing the Graphics Device 164
Let's Get Graphical 164
Color 165
Mixes 167
Color compares 167
Plane masks 167
Clipping rectangles 168
Current position 168
Back to the Drawing Board 168
Line drawing 169
Markers 173
Rectangles 175

Bit BLTs 178
Data transfer 182
Text 184
Summary 188

6. The 8514/A Hardware Interface and Register Set 189
Architecture 189
Memory 190
PC Interface 193
Lookup Table (LUT) 193
VGA Loop-Through 194
ROM 195
Registers 195
Register Specifications 197

7. Tutorial: 8514/A Registers 267
Let's Start 273
Display initialization 274
Drawing environment 279
Down to the Nitty-Gritty 283
Clearing the display 284
Line drawing 286
Long vectors 289
Short-stroke vectors 291
Bit BLTs 295
Image transfers 299
Area fills 311
Summary 312

8. 8514/A Hardware Compatibles 313
But First ... Compatibility and Existence 314
Case in point: The VGA 314
The 8514/A 315
Transparent vs. nontransparent enhancements 316
ATI Technologies 317
Chips and Technologies 320

Headland Technology....322
Western Digital Imaging (WDI)....325
Integrated Information Technologies (ITT) and Trident....326
SIS Microelectronics....327

Appendix A: Company Addresses....329

Appendix B: AI Commands by Function Number....331

Appendix C: AI-Compatible Character Definitions....333

Appendix D: IBM AFI.H....339

Appendix E: AFI Data....343

Appendix F: Listing of Hardware Registers....357

Index....361

Acknowledgments

This book would not have been possible without the efforts of the following people:

The editors and staff of *Personal Workstation* magazine published Jake Richter's 8514/A-related articles, which led directly to this book, and gave Bud Smith time away from his duties as West Coast editor to complete his contributions to the book.

Dave Wilson, Panacea Inc.'s Senior Computer Scientist, was invaluable in helping check out the nuances of the 8514/A registers.

Brett Glass read the manuscript at an early stage and pointed out many technical problems and unclear constructions.

The editors and staff at M&T Books worked under tight deadlines to get this book out in record time, cleaning up fractured syntax and figure references that led to nowhere. To the extent that this book is readable and accurate, they deserve a large part of the credit. Any errors or omissions remain the responsibility of the authors.

Dave Foley and Wes Anderson of Western Digital Imaging volunteered their technical skills toward making the book's cover look its best.

Engineers and marketers at all the companies creating 8514/A- compatible products have bent over backwards to provide us with early and detailed technical information. Working together within VESA, the Video Electronics Standards Association, they have also taken the first important steps to making the 8514/A one standard rather than many.

How to Order the Accompanying Disk

Save yourself the time and trouble of manual file entry. Contained in this optional disk are 60K worth of code that you won't have to enter yourself.

The *Graphics Programming for the 8514/A* disk contains the complete set of listings and code, as well as VESA (Video Electronics Standards Association) approved register definitions. The disk serves as a good building block for you to develop your own software for the 8514/A standard. Source code is available on 3-1/2" or 5-1/4" disk in MS/PC-DOS format.

The disk is $20, plus sales tax if you are a California resident. Order by sending a check, or credit card numbert and expiration date, to:

Graphics Programming for the 8514/A Disk
M&T Books
501 Galveston Drive
Redwood City, CA 94063

Or, you may order by calling our toll-free number between 8 A.M. and 5:00 P.M. Pacific Standard Time: 800/533-4372 (800/356-2002 in California). Ask for **Item #104-0**.

Why this Book is for You

This book has been written to fill the need for a single source of information on programming the IBM 8514/A and compatibles. The "next VGA," the 8514/A is a graphics standard from IBM that has the support of all the major PC graphics companies. Some compatibles support the Adapter Interface (AI) only, whereas others are hardware register compatible, and this book is the place you'll find information about both. We also compare and contrast the two approaches in terms of technical feasibility and marketing considerations.

In reading *Graphics Programming for the 8514/A*, you'll become familiar with the capabilities of the 8514/A and how they relate to various graphics techniques. We also discuss 8514/A-capable products from IBM, Texas Instruments, ATI, Chips and Technologies, Headland/Video Seven, Western Digital/Paradise, and more.

This book is of particular interest to the following:

- Graphics programmers. If you are involved in writing device drivers for graphics programs, this book is a must.

- Other software developers. It's important to understand the capabilities your programs will be expected to have in the near future and how to take advantage of them. This book provides that information.

- System evaluators. The stakes in PC graphics are being raised, and hardware and software with workstation-like graphics will become common. The 8514/A will lead that change, and this book gives you the detailed knowledge you need to understand it.

- End users. Everyone interested in the state of the art in PC graphics can learn from this book.

Introduction

Welcome to *Graphics Programming for the 8514/A*. This book began as a series of articles the authors wrote for *Personal Workstation* magazine — then called *MIPS* — in 1989, just as the secrets of the 8514/A were being unveiled. IBM had kept the hardware interface to the 8514/A secret since it introduced the new standard in 1987, and the first step in making it widely used was the publication of register specifications by Paradise/Western Digital Imaging, Video Seven/Headland, and Chips and Technologies. Programmers used these specifications to create the first 8514/A programs, which wrote directly to the hardware; engineers at the video companies that published them used the specifications to create 8514/A-compatible chip sets. Jake Richter's articles for *Personal Workstation* magazine were the first public description of the hardware interface and how to program to it.

Now the world of PC graphics has become more complicated. There are super-VGA boards, a growing number of 8514/A-compatible boards, and TI 34010 and 34020 graphics processor-based boards, some of which run the 8514/A Adapter Interface (AI). And rumors continue of an 8515/A from IBM that will extend the 8514/A standard to new heights.

The 8514/A standard itself has a lot of parts. There's the Adapter Interface (AI), originally published by IBM. The AI is supported not only by all currently available 8514/A-hardware-compatible chip sets but also by many 34010- and 34020-based boards that also run TIGA (Texas Instruments Graphics Architecture). It's also rumored to be compatible with future generations of the 8514/A from IBM. This book documents and explains in detail how to use the AI for readers who want the widest possible compatibility.

However, writing directly to the registers is about twice as fast as going to the AI. The 8514/A is only being taken seriously as a standard now that hardware compatibles promise to expand the market beyond IBM's own moderately successful effort. There are reasons, though, for software developers to write drivers for the AI, directly to the registers, or both. This book explains and demonstrates, through reference sections and code samples, how to do so.

The problem, of course, is that there are a number of manufacturers of 8514/A-compatible chip sets, from IBM to Integrated Information Technologies. Every chip set endeavors to support a compatible set of registers and programming commands. Each chip set also has its own proprietary extensions, which provide differing features. In the middle part of this book, we describe how to program to the compatible 8514/A register set. Later, we devote a chapter to the current manufacturers and their products and the proprietary extensions they include so readers can compare and contrast the different feature sets. We also provide a brief description of the TI 34010 and 34020 and the TIGA standard for those who might want to consider using this extensible graphics interface. This way, readers can write programs that run on all 8514/A register compatibles or chip-set-specific programs that take advantage of a given set of features.

We also describe the documentation and sample code provided by each manufacturer. In performing an 8514/A development project, you will probably want to get this information from at least one manufacturer, and it may affect which product you choose to work with. You may also want to use this information to help you decide which chip set to choose for chip-set-specific programs.

This book would have been much harder to write without the work of the VESA (Video Electronic Standards Association) committee and its 8514/A subcommittee. This group has agreed upon several important elements of the 8514/A standard, including a standard way to pass through VGA signals and, most importantly for our purposes, a standard set of register names for the compatible register set.

There has been hope that the members would also agree on a compatible set of extensions to the IBM hardware standard, but at this writing, all extensions are proprietary to a particular chip set and might change across a manufacturer's product line or even within a given model over time. So an important purpose of this book is to record the work of the VESA 8514/A subcommittee and to provide you with a survival guide for working in those areas in which the subcommittee has not yet agreed to tread.

The final major purpose of this book is to serve as a clearinghouse for additional information. For example, we can document the differences between chip sets to give you a way to decide which one to use, but in the final stages of making that decision, you'll want to get information from one or more manufacturers. We provide contact names and phone numbers for each company in the market and a way to contact us via electronic mail in case you have any questions.

We hope that by gathering all this information in one place and by providing instructions, examples, and reference material, we'll help to popularize the 8514/A standard — both the Adapter Interface and the compatible register set. Having this information available should help new, better products get to market more quickly and with a better chance of being bug-free.

Who You Are

This book is written primarily for experienced graphics programmers who have used the C language and who want to write drivers for the 8514/A. Most readers will have experience programming the EGA and/or VGA graphics standards. Others will be working on PCs for the first time, having started in the workstation world or other parts of computing, and finding the 8514/A to be the first PC graphics standard worth writing for. In either case, you will find using the 8514/A to be like using a fairly well written graphics library.

This book is of most use to those writing drivers for DOS programs. In many cases, those writing for Windows or Presentation Manager will simply use the graphics-programming routines provided as part of the graphical user interface.

Novice graphics programmers can also learn a lot from this book. However, we don't explain all the basic concepts underlying graphics programming and don't provide basic tutorials for those just getting up to speed. But you can go a long way by reading this book, getting the sample code available and experimenting with it. There are other books that cover graphics-programming basics; we particularly recommend Roger Stevens' best-selling *Graphics Programming in C* (M&T Books, 1988) as an excellent source of both how to and why.

We also hope to reach a small group that used to be more prevalent in the early days of personal computers but that has not yet faded away entirely: intelligent laymen who don't do a lot of programming but who read books such as this to get a deeper understanding of the products available and their capabilities and limitations. We don't have space in this book to define all our terms or explain the basics of graphics programming, but you can learn a lot through common sense and immersing yourself in information. This book explains not only what the differences between the various 8514/A-compatible products are but also what those differences mean.

Who We Are

Both of us are big fans of making the best possible graphics available to the greatest possible number of people, and we believe the 8514/A is an important step in making that happen. The 8514/A and the TI 34010 and 34020 are making the necessary hardware available to the public, but software must take advantage of the new standards in order to make them successful — thus this book.

The authors have worked together on several smaller writing projects, mostly for *Personal Workstation* magazine. Jake Richter is the lead author of this book. He was originally a principal engineer at Number Nine Corporation, working on the Pepper series of boards, including the Pepper SGT with both Intel 82786 and TI 34010 graphics chips; NNIOS (the Number Nine Intelligent Operating System); and other projects. He is now President of Panacea Inc., a graphics consulting firm in Londonderry, New Hampshire. Panacea has published the first program to write directly to the 8514/A registers — DLD 8514/A, which is a display-list driver for AutoCAD. This driver works with AutoCAD's built-in display-list capability to translate the object format in which AutoCAD normally stores objects — and which it must translate every time it updates the screen — to a display list that occupies more space in memory but is far faster to put up on the screen. DLD 8514/A gets an additional speed advantage by writing directly to the 8514/A registers.

Panacea has written drivers for many of the companies that have created 8514/A-compatible chip sets and for companies using the TI 34010 and 34020 as well and continues to create custom drivers for a variety of clients. It was also the first software company to join the formerly hardware-only VESA committee and is currently an active member of the 8514/A subcommittee. Jake has had a voice in setting the standards currently agreed upon by all 8514/A chip-set makers and in proposing a new generation of standards to carry the 8514/A market forward.

Jake is a contributing editor of *Personal Workstation* and writes the graphics column and graphics articles on a regular basis. He's also a frequent contributor to *PC Week*, as a writer on the opinions page and a source for comment on graphics topics.

Bud Smith has written some of the book and edited the rest of it, providing the detailed outline and writing the first few chapters and several of the appendices. His background includes several years of programming on minicomputers and micros, including a year of Apple

II assembly-language programming. He has written several books, most notably *Programming the Intel 80386*, published by Scott, Foresman, January 1987.

In his role as West Coast technical editor for *Personal Workstation* magazine, Bud has covered the convergence of high-end microcomputers and low to midrange workstations, with special attention on microprocessors, graphics, and desktop video.

How the Book Is Organized

This book can serve both as an instructional text and as a reference for those who need a specific piece of information in a hurry. We have divided it into several parts to make finding what you need as easy as possible.

Part I describes the 8514/A and its history, the Adapter Interface, and the register set. It lists the compatible chip sets and gives an overview of their features and describes software development for the AI and for the registers.

Part II, the how-to part, includes reference and descriptive information for programming the AI and the registers. Two chapters — one for the AI and one for the registers — are tutorials that describe how to get from a given goal to a working program.

Part III describes the different 8514/A-compatible chip sets available, their features, their extensions, and the programming extensions offered by each manufacturer. These chapters also contain all the information we can get our hands on about incompatibilities, special features, and programming enhancements, plus a description of the documentation, tools, and support available for developers working with a given chip set. This section should be of great value to those trying to get the most from their programs while keeping them compatible with the whole 8514/A-compatible world.

Part IV is pure reference — appendices and index. The appendices include AI functions and register lists and sources for further information. This section will hopefully be well-thumbed-through as you use the book.

In addition to the book itself, we've created a supplemental disk, available from M&T and described below.

Tools and Resources

This book includes program examples in C. We used Microsoft C Version 5.1, but the code provided here should work with other popular C compilers too. We are also making available a disk containing sample code. Sample programs include all code described in the tutorials in chapters 5 and 7.

A great deal of information is also available from the manufacturers of 8514/A-compatible products. Chapter 8 provides a description of the chip sets available and their supporting documentation and code libraries.

Keeping Updated

This book is being written at a time when the first 8514/A-compatible chip set, in the Paradise 8514/A from Western Digital, is just going on sale and chip sets from Chips and Technologies and Integrated Information Technologies have been demonstrated in public for the first time. Other manufacturers are planning to introduce products over the next several months and have generously given us advance information about their products. This book would not have been possible without their cooperation.

We are making every effort to provide information that is as complete and accurate as possible, but of course we won't be able to in-

clude changes that occur after the book goes to print. We believe that all manufacturers will make a point of supporting the compatible subset of registers we describe here, so programs written to work on all 8514/A compatibles will do so. What is certain to change is the set of extensions offered by each manufacturer. If you are planning to write programs that take advantage of features specific to a given board, contact the manufacturer now at the address given in Appendix A to ensure the features you want are indeed available and to learn about new features and products.

Readers who send in the registration card will be notified of upcoming revisions to this book and can subscribe to the Panacea newsletter. We are also available on-line to discuss questions arising from the book and to provide new information. Our BIX names are jrichter and budsmith. Jake is also on MCI Mail as jrichter and CompuServe as 75130,2705. When TechNet, M&T's on-line service, is up and running, you will also be able to contact us both there.

♦

Portions of this book have appeared in substantially different form as articles by Jake Richter and by Jake Richter and Bud Smith in *Personal Workstation* magazine during 1989 and 1990.

1

PC Graphics Standards

This chapter describes the currently popular PC graphics standards and examines how the 8514/A fits in. Although this material is not strictly necessary for writing an 8514/A driver, we feel that programmers are increasingly making marketing decisions about what platforms to support and in doing so are being forced to choose between a wide and wild variety of hardware and software options. We offer this chapter to provide some context for the technical material that makes up most of the book. We'll begin with a look at existing PC graphics standards.

With one exception, the adoption of new PC display standards has followed a simple model:

1. IBM comes out with a new graphics board.
2. Other manufacturers successfully imitate, or clone, it.
3. Programmers write drivers for their applications to allow them to take advantage of the new standard.
4. Hardware manufacturers (not IBM) create extensions to the standard to satisfy customer requirements.
5. IBM comes out with a new graphics board that includes some of the features that had been provided as extensions but that may be incompatible with the previous generation.

All widely used PC graphics standards except one have followed this pattern. The exception is the Hercules Graphics Card, which was the first board to offer monochrome-graphics support for PCs. Although it originated outside IBM, the Hercules card has also followed the steps

listed above, and its functions have been imitated and built in to a variety of add-in cards and motherboards.

The PC graphics standards that have been successful to date are:

• Monochrome Display Adapter (MDA), which displays 25 lines by 80 characters. The MDA (and CGA, EGA, and VGA text modes) sends 2 bytes per character to the screen. The first byte is the character's ASCII code; the second is an attribute byte that determines whether the character blinks, displays in boldface, and so on. Only 4,000 bytes (2,000 characters with one attribute byte each) need to be sent to fill an entire screen. The MDA displays characters in a large 9 x 14-pixel box and, until the introduction of the VGA, provided the only really good character resolution of any IBM display standard.

• Color Graphics Adapter (CGA), which displays 320 pixels across and 200 pixels top to bottom in its four-color graphics mode and 640 pixels across by 200 deep in monochrome and text modes. This means that characters are 8 x 8 in text mode and hard to read.

• Hercules Graphics Card (HGC, or MGA for Monochrome Graphics Adapter), which is based on the same CRT controller as the CGA and MDA but is much more capable. HGC gives 720 x 348-pixel resolution in monochrome graphics mode and 720 x 350-pixel resolution in character mode, allowing sharp 9 x 14 characters like those available under MDA.

• Enhanced Graphics Adapter (EGA), with 640 x 350-pixel resolution and 16 colors out of 64 in graphics and text modes. This allows a sharp 8 x 14 character in text mode.

• Video Graphics Array (VGA), which has 640 x 480-pixel graphics resolution with up to 16 colors and 720 x 400-pixel resolution for text. This allows a 9 x 16 character.

Each IBM standard has nearly complete compatibility with the ones before, so VGA can run MDA, CGA, or EGA programs with reasonable success. This compatibility has greatly eased the burden on users by allowing their old programs to run on new hardware, but it has had the unfortunate side effect of making it easy for application developers to avoid creating new drivers that take advantage of improved standards.

Although VGA does not have much better resolution than EGA, it does mean a big change in monitors: from digital to analog. An EGA monitor has six color lines connected to it — red, red' (red prime), blue, blue', green, and green'. Because any of the six lines can be on or off, $2^6 = 64$ possible colors can be sent to the EGA monitor. Each color can have any of four values, based on the use of two lines that can be on, off, or one on and the other off. However, due to the design of the EGA, only 16 of the 64 colors are available at any one time.

A VGA monitor is connected to three color lines — red, green, and blue. But each line is an analog connection, which can carry a wide range of voltages. IBM's original VGA cards have a 6-bit DAC (digital-to-analog converter) for each gun, meaning any of 64 possible digital values can be fed into the analog line for each gun. This yields $64^3 = 262{,}144$ possible colors. The amount of memory limits the number of colors that can be on-screen at any one time — 4 bits per pixel means 16 colors can be displayed simultaneously from the palette of 262,144 and 8 bits per pixel allows 256 colors.

The VGA standard is the first to provide good graphics — some images that use 256 colors are quite striking — and readable text with good performance, and it's been very successful. A VGA add-in card or even VGA capability on the motherboard, as in IBM's PS/2 line, has become a lowest common denominator for business computers. Many drivers have been written for VGA, most simply by modifying extended-EGA drivers that already existed.

Many attempts have been made to create an extended-VGA standard at an 800 x 600 resolution with 16 colors, and some VGA cards go up to 1,024 x 768 by 256 colors with an upgrade to 1 Mbyte of RAM. The problem is that every board manufacturer has created a different way to get to these higher resolutions. This means that each board must come with its own set of drivers and that any program or graphical interface that doesn't have a driver included with the board can't be used at the extended resolutions. Most boards include drivers for Windows, AutoCAD, and Lotus 1-2-3, with some also supporting Presentation Manager, Microsoft Word, WordPerfect, and/or GEM; DOS programs not represented in the driver set can't use the higher resolutions.

All IBM-standard graphics boards through extended VGA are "dumb" frame buffers — they provide a memory area for the system's main processor to draw in and then take the image from memory and convert it to a screen display. Different boards may have more memory or be more efficient about getting data on-board or moving it to the screen, but all the boards depend on the system CPU to get the image on-screen. This is not really so bad; high-end system processors have improved about as quickly as graphics standards. VGA has up to 4 times as many bits per pixel as CGA and almost 5 times as many pixels, for a total throughput about 20 times that of CGA. Processors have improved at least that much too: A 25-MHz 386 is about 40 times more powerful than a 4.77-MHz 8088, and a 25-MHz 486 is about 3 times more powerful than that.

The Need for a Better Standard

A lot of high-resolution graphics boards fit specific needs for a given level of price and performance. Many of these are based on the TI 34010 and 34020; others are based on other graphics chips or proprietary silicon made by the board's manufacturer. Because the boards and the monitors that go with them are specialty products and are on

the cutting edge of technology, they sell in small quantities and are consequently expensive (the boards cost well over $1,000).

The PC world is based on the widespread adoption of standards. Most are de facto standards based on imitating what IBM has done; others come from manufacturers, working either singly or in groups, or are the result of visionary work by various standards committees. (Most of the work of standards committees consists of codifying marketplace standards; only a few, such as SCSI-II, involve much new work.)

The VGA standard has been tremendously successful, and even at the basic resolution of 640 x 480, with 16 colors, it's quite capable. However, three main applications are driving graphics beyond VGA:

• Multitasking. Users running several tasks want to have a main window of 640 x 480 and still have room for icons and pieces of other windows on-screen.

• CAD. The closer a screen image is to the real-world object it models and the faster the image is updated, the more productive the user and the more useful the image. The rule here is "more is better" — more colors, more screen area, more resolution, and more speed.

• Desktop publishing. Although the rule in CAD is more is better, there are quantum steps in the usefulness of desktop publishing improvements. A one-page display is useful, a two-page display is much more useful, and a one-and-a-half-page display is not necessarily useful. And monochrome is plenty good enough for many desktop publishing users. Refresh rate, which makes small type easier to stare at for hours on end, may be more important than small improvements in screen-updating speed. For instance, compared to the standard updating speed of 60 frames per second, 70 frames per second flickers less; many European governments are demanding this as a minimum standard for certain monitors.

There are already specialized solutions for each of these dedicated tasks. Heavily multitasking users tend to use workstations, with 16- or 19-inch screens. CAD users run a variety of systems and tend to run the highest-end graphics available for their systems. Desktop publishers often use the Mac, which easily supports a variety of screen solutions and is leading the way in affordable support for 24-bit color.

To be anything other than a specialized solution, a new mass-market PC standard has to support users from all the areas listed above and must also be cheap. It's not enough for the graphics board to be affordable; it must also work with monitors that are currently in the midrange of price and that can be quickly driven lower in price.

Another important rule for any new standard is that it must do everything the previous standard did, plus offer additional features. For example, a new standard that offers higher resolution but that updates the screen more slowly than the previous standard isn't likely to become popular.

The AI vs. Hardware Compatibility

When IBM introduced the 8514/A in 1987, it didn't publish the register specification but only the Adapter Interface (AI) — a software library that interfaces with the registers. The trouble is that the AI must check all the parameters passed to it for reasonableness and do other "responsible" things, and it therefore runs about half as fast as tighter routines would. Even with this additional overhead, performance was good — better than the less-capable VGA and competitive with TI's 34010.

The graphics industry has now reverse-engineered the 8514/A register specification and is coming out with hardware compatibles that support programs that write to the AI and programs that write to the registers. VESA and its 8514/A subcommittee are working together to ensure that the reverse-engineered products are compatible in the way

they pass through VGA signals and in their register interface. This gives software developers a stable target to write to so that a single program can run on all 8514/A compatibles.

The AI is still important because IBM has promised that the long-awaited successor to the 8514/A will be AI compatible but not 8514/A register compatible and because TI 340X0-based board manufacturers have included AI support as an option, described below. IBM has recently introduced a new, fast graphics board — the Image Adapter/A — which has capabilities similar to the 34010's, but it is keeping quiet about this board for now. The Image Adapter/A is 8514/A-AI compatible but not register compatible. Therefore software written to the AI will run not only on 8514/A hardware compatibles but also on current and future IBM graphics boards and some TI 340X0-based boards. However, many software applications will write directly to the hardware in order to get the best possible performance. Others will write to the AI for compatibility, and some products may have both a register-level and an AI version.

How the 8514/A Fits In

To go much beyond 640 x 480 requires a graphics board with its own graphics-processing power; otherwise, the CPU is too tied up doing graphics and performance falls rapidly. A graphics processor saves CPU time in that it takes high-level commands from the CPU, such as "draw a line between these two points," and executes them itself; a frame buffer forces the CPU to draw each pixel in the line by itself.

The 8514/A is a low-end solution; it uses a coprocessor, which accepts commands one at a time from the CPU. A true graphics processor such as the TI 34010 can execute whole programs on its own without intervention from the CPU, but it needs a separate memory area in addition to display memory for program storage. This adds cost to the board and adds a new area for vendors to distinguish themselves, making the boards distinct and a mass-market standard more diffi-

cult. TI's recently introduced Business Graphics Array (BGA) chip solves some of these problems, but it's too soon to tell how widely used it will be.

The 8514/A is enough of a step up that it warrants specific attention from users and programmers. At 1,024 x 768, the 8514/A offers two and a half times as many pixels as standard VGA does. However, IBM's standard 8514/A uses an interlaced display, causing flickering, and in its low-end configuration supports only 512K of video memory, which can provide just 16 colors. We believe users will insist on noninterlaced displays and a full megabyte of memory to give 256-color support at 1,024 x 768 and will insist on programs that take specific advantage of the 8514/A.

The graphics world is getting easier for users and programmers. Many programs run under Windows, which has drivers for almost all major graphics boards. But for the next few years, most users will still have at least some DOS-specific programs, so for this period many application-specific drivers will be needed.

In its high-end configuration, with a board that supports noninterlaced display and has 1 Mbyte of video memory to show 256 colors at 1,024 x 768, the 8514/A does everything VGA does and then some — more resolution, more speed, no flicker, and more colors. And under Windows and Presentation Manager, driver support is already available, though it will be more chancy for DOS programs.

Market Prospects for the 8514/A

The PC market has proven very willing to move up to new IBM-derived graphics standards once boards and monitors are available at a reasonable price. For the next step beyond VGA, "at a reasonable price" has been widely seen as less than $1,000 for a graphics card and less than $1,000 for a 1,024 x 768 noninterlaced monitor.

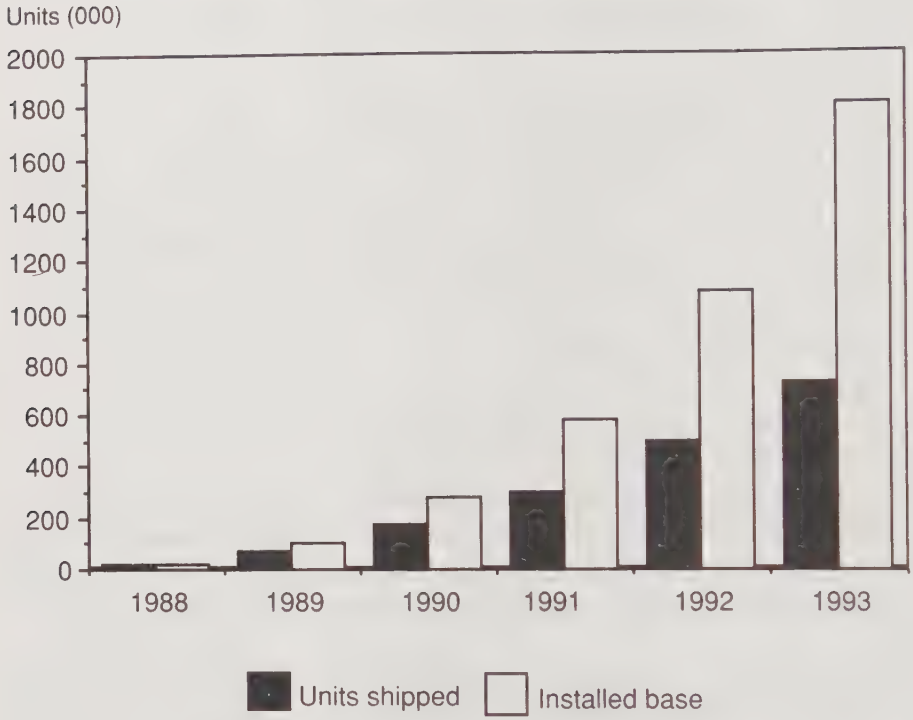
The first 8514/A product introductions feature boards costing less than $1,000 (street prices) with a full megabyte of video RAM. Lesser products with only half a megabyte of memory and/or DRAM in place of VRAM are coming in at between $500 and $750. (Some benchmark figures indicate that page-mode DRAMs offer performance 90% as good as that of VRAM, at less than half the cost.) Monitor costs for 14-inch 1,024 x 768 noninterlaced displays, such as the Sony Multiscan HG, are also coming in at less than $1,000. And all this is occurring at the very start of the 8514/A-compatible market, indicating that board prices will be very affordable — perhaps as low as $300 street price — within one or two years.

The growth in graphical user interfaces is also a big plus for the 8514/A. Windows and Presentation Manager users need only one driver for their board in order to run all the programs that use the graphical environment. Although DOS will continue to be important for years to come and individual 8514/A drivers will be needed for DOS programs, an important part of the market will be insulated from this concern.

The final boost for 8514/A is the product differentiation that is occurring among vendors, even as the first boards appear. Some boards are 8514/A only, others include VGA compatibility, and yet others do both in a chip set that can easily be integrated on the motherboard. VRAM and DRAM models with 512K and 1-Mbyte options are being offered by most vendors, and some are even adding custom hardware to support features such as anti-aliasing for text or vectors, without a loss of compatibility. This means that buyers who are most concerned with price or with specific kinds of functionality are very likely to find a product to meet their needs.

TI-based boards are making a strong bid for a share of the high-resolution market, as described below. But PC market researchers and others foresee the larger share of the market going to the 8514/A. Figure 1-1 shows Dataquest's projections for the future growth of the 8514/A. Although these numbers are definitely not set in concrete, they do show the industry's optimism about the 8514/A's prospects.

Figure 1-1: IDC projections for 8514/A market growth

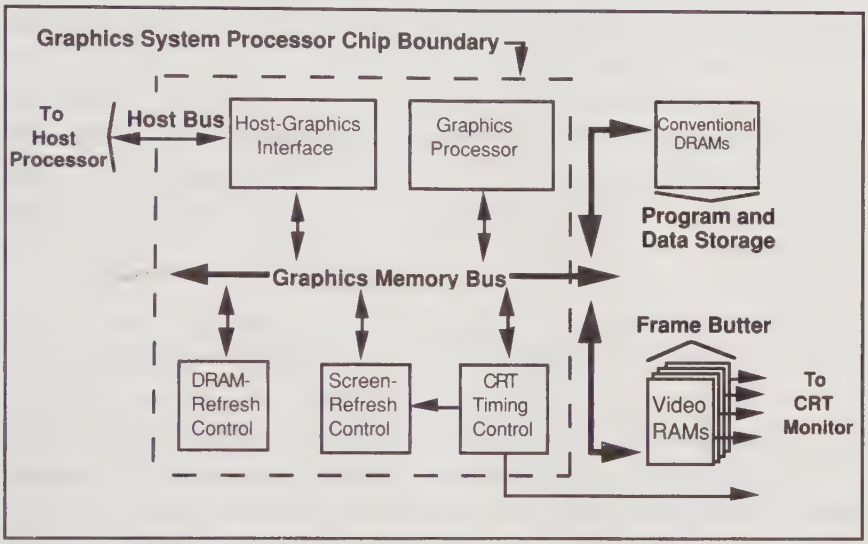


(Source: International Data Corp., 1989)

The TI 34010 and 34020

The TI 34010 and 34020 are true graphics processors, with their own program memory in addition to the video memory used to hold the currently displayed screen image (see Figure 1-2). Being true processors, the TI chips can execute entire programs without intervention from the CPU. Of course, a graphics chip that does its own thing totally is generally not much use, so the usual model is for the CPU to download graphics routines — to draw a particular shape efficiently or put up a window — to the video board's program memory and then direct the graphics chip to run individual routines as needed. This makes the graphics card quickly customizable to any task.

Figure 1-2: Block diagram of the TI 34010's major functions and interfaces to devices



In addition to a large number of proprietary interfaces, the TI 34010 and 34020 run a software interface called TIGA (Texas Instruments Graphics Architecture), which is much like the 8514/A's AI. It's ironic that the 8514/A didn't take off until programmers could write to the hardware, whereas TI is banking on a software interface to make its chips more popular. One small reason for the TIGA interface is to shield programs from the differences between the 34010 and 34020; code written to the 34010 hardware is not guaranteed to run on the 34020 unless several precautions are taken.

TI is taking other steps to make its chips an 8514/A-type standard. It has introduced the BGA (Business Graphics Array), a chip that combines several "glue logic" chips that were needed to build a 34010 board. The BGA also eliminates the need for separate program memory by allowing the program memory space to be mapped onto unused

video memory — so a 1-Mbyte board might use just under 800K to support the screen image and 128K to hold program code. This makes it easier to build a 34010 board for less than $1,000.

Several factors remain that will tend to make prices of 34010-based boards higher than those of 8514/A boards. There is still a higher-end orientation to the 34010, and almost all boards based on it will probably have a full 1 Mbyte of memory and use VRAM rather than offering a DRAM version as an option. Under-$1,000 34010-based boards will probably be slower to offer differentiating features, such as anti-aliasing and a wide range of drivers, than under-$1,000 8514/A boards.

Of the 34010-based board makers, only Hercules has announced a board that combines VGA and the 34010 on a single board, though not in a single chip as with the Integrated Information/Trident solution. And to the extent that the 8514/A is accepted quickly, its producers will achieve great cost efficiencies, which will drive prices down quickly and so further increase user acceptance.

The TI 34010 does have definite advantages. The existence of the 34020 means that 34010 users have a ready-made upgrade path if they need more performance (CAD users and others always need more performance). However, generally only code that writes to TIGA runs transparently on both chips. And 34020-based boards will bear a price premium because the initial version of the BGA works only with the 34010; 34020-based boards will still need additional glue logic and separate program memory, not to mention that 34020s cost substantially more than 34010s.

The 34010 and 34020 are much more flexible than the 8514/A because they are full graphics processors. Improved downloadable graphics routines may speed the 34020 well beyond the 8514/A's capability, especially for highly demanding applications running under DOS. However, graphical user interfaces that support multitasking usually control the graphics board. Such interfaces can be supported by spe-

cific downloaded code to support windowing and so on, but no application-specific code can then be downloaded, preventing applications from taking full advantage of the board.

The high performance of TI-based boards, their acceptance as a high-end standard, and the availability of an easy upgrade path from the 34010 to the 34020 all ensure that TI will be an important part of the graphics market. TI-based boards may form the launching point for the next graphics standard after the 8514/A, especially as monitors capable of 1,280 x 1,024 and even 1,600 x 1,200 resolution drop in price. But it seems very possible that the 8514/A will have a larger share of the market for at least the first half of the 1990s.

Benchmarking 8514/A and 34010 Boards

Benchmarking 8514/A boards by using a given set of tests is a useful exercise that tells you a lot about the boards' performance, and therefore their value, although the tests won't tell you about added-value features such as anti-aliasing capabilities or the number of drivers available. Such tests can also be valuable for 34010-based boards, although these may have even more differing features than 8514/A compatibles. But using benchmarks to compare 8514/A compatibles with 34010-based boards is hazardous.

The first pitfall is that most comparisons are apples-with-oranges tests designed to favor one side rather than the other. The kind of drivers used for testing, for example, is important. AutoCAD performance is one popular and useful metric, but many tests compare an 8514/A driver that writes to the AI and does not use display lists with a 34010 driver that bypasses regular TIGA functions and does use display lists. Other low-level comparisons write directly to the 8514/A's hardware on the one hand but use TIGA on the other — and don't take advantage of all TIGA features to boot. When looking at benchmark results, make sure that the same parameters apply to both sides of the comparison.

It does seem that many users of 8514/A compatibles will be running software drivers that write directly to the hardware, whereas TI will emphasize products that use TIGA drivers for compatibility across 34010- and 34020-based boards. So it may be that the best real-world comparison will be between an 8514/A board running software that writes to the registers and a 34010 board with software that uses TIGA.

Three factors affect overall drawing performance:

- The amount of CPU time used to manage the graphics board.

- The time it takes to get commands and data to the graphics board, calculate initial data, and start drawing the first pixel, called the setup time.

- The time to write the pixels, called the pixel drawing speed.

Both the 8514/A and the 34010 off-load most graphics work from the CPU. With VGA and earlier PC graphics standards, the CPU executed all the graphics commands and actually wrote each pixel into memory, slowing program execution. With the 8514/A, the CPU simply sends low-level commands, some as simple as "draw a line from the current position to (x1, y1)," to the graphics board, which then executes the command. The 34010 goes a step further; a single command such as "set up a window with vertices (x1, y1) and (x2, y2)" can cause a number of lines to be drawn if the right routines are in the board's program memory.

Because the 34010 can take higher-level commands, it also has an advantage in the amount of traffic that must go over the bus. However, the difference in bus-traffic requirements between any frame buffer (such as VGA) and any controller (such as 8514/A or 34010) is much greater than the difference between individual controllers.

Pixel drawing speed is an area of great controversy in which many unfair comparisons are made. Both sides seem to agree that the 34010 is faster for pattern filling, an important operation in windowing environments, and that the 8514/A is faster in line drawing, the most important single operation in CAD. But how these two primitive operations, and others on which benchmarking results differ, translate to application performance is controversial.

It may be fair to say that for most applications, the 34010 and 8514/A provide similar performance. The 8514/A will probably have a price advantage, and the 34020 provides the 34010 with a better growth path. Driver quality can outweigh low-level hardware performance in the results users get, and this will be an area of great competition among all the different boards.

Memory Math

It's important for programmers to know how memory is used on 8514/A boards. Table 1-1 shows the amount of memory used and the amount remaining for working space at different resolutions and color counts on 512K and 1-Mbyte boards. A resolution of 1,280 x 1,024 is included because it's available as an extension on many 8514/A hardware compatibles.

A complication caused by current VRAM technology affects many boards on the market today. Although the screen might be only 640 pixels across, memory is tied up as if the line were 1,024 pixels across because memory can be committed only in chunks that are powers of 2. This decreases the amount of working memory available to the programmer. Although some newer memory implementations avoid this problem, most existing boards are affected by it, and programmers can count only on the smaller amount of memory being available.

Resolutions and numbers of colors used cover standard VGA (640 x 480), standard 8514/A (1,024 x 768), and the resolution used by many

34010/34020 boards and what we'll call super-8514/A boards (1,280 x 1,024). The number of colors available on a given board is limited by the design of the board and the amount of memory installed.

Table 1-1: Memory available and used on 512K and 1-Mbyte boards

		Available (K)		
Resolution/Colors	Used (K)	256K on board	512K on board	1 Mbyte on board
640 x 480 monochrome	60*	196	452	964
640 x 480 by 4 colors	120*	136	392	904
640 x 480 by 16 colors[1]	240*	16	272	784
640 x 480 by 256 colors[1]	480*	N/A	32	544
1,024 x 768 monochrome	96	160	416	928
1,024 x 768 by 4 colors	192	64	320	832
1,024 x 768 by 16 colors[1]	384	N/A	128	640
1,024 x 768 by 256 colors[1]	768	N/A	N/A	256
1,280 x 1,024 monochrome	256**	0	256	768
1,280 x 1,024 by 4 colors	512**	N/A	0	512
1,280 x 1,024 by 16 colors	1,024**	N/A	N/A	0
1,280 x 1,024 by 256 colors	2,048**	N/A	N/A	N/A

[1] 8514/A-compatible resolutions

* 640 pixels treated as 1,024 pixels by older memory implementations

** 1,280 pixels treated as 2,048 pixels by older memory implementations

The figures in Table 1-1 apply to VGA and TI 34010-based boards too; they're derived by rounding the horizontal resolution up to the next power of 2, multiplying by the vertical resolution, dividing by the number of bits per pixel, and then dividing by 1,024 to convert the resulting figure to kilobytes. VGA boards do not suffer from the power of 2 limitations.

The table not only shows how much working space is available on a 512K or 1-Mbyte board at a given resolution and color count, but it also shows what resolutions and color counts aren't supported by a given amount of memory. Only one standard 8514/A resolution — 1,024 x 768 by 256 colors — is unavailable on 512K boards; this resolution may be ignored by many programs if program designers don't want to sense the amount of memory installed and run different code depending on the result.

Double buffering is used in animation and special effects. Boards that support double buffering display from one block of video memory while the next image is built up in another block; once the new image is complete, the program can then quickly change a pointer to make the board display from the new block. Double buffering is limited on the standard IBM 8514/A, however; there's no way to quickly switch the block of memory you're displaying. The only way to simulate this on the standard IBM 8514/A (and all the compatibles we know of) is to display at a resolution that uses up half or less of the total video memory, build up the next image in the remaining available memory, and then BitBLT the next image into the displayed memory area (or use the palette to switch between two 16 color images).

Table 1-1 shows that a 512K 8514/A board can hold two 640 x 480 images with up to 16 colors. (At this resolution and number of colors, the amount of available video memory is greater than the amount used by the displayed image.) A 1-Mbyte board can hold two 640 x 480 images with up to 256 colors or two 1,024 x 768 images with up to 16 colors, although the relative slowness of transferring bit blocks may lead you to want to use lower resolutions and/or fewer colors than video memory can actually hold.

Monitor Requirements

The acceptance of new graphics standards is very much dependent on the availability of suitable monitors at reasonable prices. The key factors in making a monitor that sells for less than $1,000 using today's technology are:

• Keeping the monitor size down — to 16 inches or less. Larger monitors are much heavier and more expensive; the smaller a monitor at a given resolution, the smaller its pixels must be.

• Not forcing the pixel size, or dot pitch, to be too small; 0.25 to 0.31 mm is achievable and supports 8514/A resolutions on a 14- to 16-inch screen. A much smaller dot pitch costs more.

• Keeping the maximum horizontal scan rate down. But the lower the horizontal scan rate, the lower the resolution that can be supported without interlacing. The horizontal scan rate determines the number of horizontal lines a monitor can support, and monitors are often classed as "480 line" or "768 line," depending on the number of noninterlaced lines they can display at 60 frames/second.

• Supporting as few scan rates as possible. All early PC monitors and many current ones support only one scan rate and won't work or can be damaged if hooked to a graphics card running at a different rate. Monitors for 8514/A need to support at least two scan rates — one for 480 lines (640 x 480) and another for 768 lines (1,024 x 768). Most are multisynchronous across a wide range of scan rates.

• Certification considerations. All monitors sold to business are FCC Class A certified, but these monitors can't be sold for home use. Monitors sold for use at home must pass the more stringent Class B certification; virtually no 1,024 X 768 monitors available at this writing pass Class B.

In 1987, when the 8514/A was introduced, a 768-line monitor was a rare and expensive beast. IBM developed the 8514/A board in tandem with a reasonably priced, manufacturable monitor — the 8514. Displaying 1,024 x 768 resolution with its relatively large 0.31-mm-pixel pitch means that the monitor must measure 16 inches diagonally. The 8514 is a dual-sync monitor that can run at either 31.5- or 35.5-kHz horizontal scan rate. It supports either 480 display lines at 60 updates per second (31.5 kHz) or 768 lines, interlaced, at 43 updates per second (35.5 kHz) — the numbers are worked out in the section entitled "Monitor Math" below.

The worst thing about the IBM monitor is that it's interlaced at 1,024 x 768 resolution. Interlacing is a technique that allows a low-capability monitor to display more lines. When interlacing its display, the monitor paints a screen image in two passes — first putting up the even-numbered lines and then going back and filling in the odd-numbered ones. This won't work if the phosphors that were lit up on the first pass fade too quickly, so long-persistence phosphors are used. Short-persistence phosphors cause flicker as the first half of an image drops away before the second half gets on-screen; long-persistence phosphors cause streaking because the pixels from one screen image are still lit when new pixels are being put on-screen. IBM did a good job of picking phosphors — the 8514 flickers some and is a little fuzzy but not extremely so in either direction.

Interlacing is unacceptable for CAD work, which uses a lot of thin horizontal lines. One-pixel-high horizontal lines fade too much between refreshes; three-pixel-high lines "strobe" disturbingly as first the middle row of pixels and then the two outside rows are redrawn. Natural images such as faces and even on-screen windows can look good enough on an interlaced display; however, users are increasingly resisting interlacing.

Monitor Math

It's relatively easy to derive the number of lines a monitor can display from its scan rate or to convert a given scan rate to the number of displayable lines. This technique is useful for anyone who wants to have a better understanding of the hardware issues that underlie high-resolution graphics.

To display 768 lines, noninterlaced, at 60 times a second (just enough times to provide a good image), the monitor has to draw 768 x 60 = 46,080 lines per second, a horizontal scan rate of 46 kHz. Another 5 to 10% margin is needed for overhead such as synchronization signals, so a 768-line monitor is usually rated at about 50 kHz. Therefore, only monitors rated at about 50 kHz or greater can display 1,024 x 768, noninterlaced.

To display the next popular resolution — 1,280 x 1,024, noninterlaced, at 60 times per second — the monitor has to draw 1,024 x 60 = 61,440 lines per second, giving 63.5 or 64 kHz with synchronization overhead. And 1,600 x 1,200 noninterlaced requires 1,200 x 60 = 72 kHz, which is usually rounded to 74 to 76 kHz. As the progression of numbers — 50 kHz to 64 kHz to 76 kHz — shows, the monitor frequency increases quickly with increasing resolution.

These numbers are varied in several ways to provide differing levels of capability. When IBM wanted to use an interlaced display (lines are drawn alternately, first odd and then even), it had to adjust the frame rate. If only 60 of these half-screens were put up per second, the display would be unacceptably sparse, so IBM increased the rate to 86 half-screens, or 43 full screens, per second. This gives a horizontal scan rate of 768 x 43 = 33,024 lines per second, usually rated as 35.5 kHz, giving an acceptable image without going to the full 50 kHz needed for noninterlaced display.

Some people want more performance rather than less out of a given spec. Many users, especially desktop publishers, find staring at a dis-

play with 60 frames per second tiring. More and more monitor manufacturers are upping the display rate to 66, 70, or 72 frames per second, with a proportional increase in the monitor's scan rate at a given number of lines. This increase makes the image crisper and noticeably easier to look at for long periods; European standards enforce a minimum rate of 70 frames per second.

IBM's 8514 monitor supports 480 lines at 60 frames per second (noninterlaced) and 768 lines at 43 frames per second (interlaced), so it needs to synchronize to two horizontal scan rates — 31.5 kHz and 35.5 kHz. This dual-synching monitor is much cheaper than multiscanning monitors that can handle a wider range of scan rates. But it prevents support of other resolutions such as 800 x 600, let alone 1,280 x 1,024, and these resolutions are not supported under the 8514/A standard.

The PC Graphics Industry

Although IBM sets most PC graphics standards, it's the rest of the PC graphics industry that makes a given standard more or less successful. IBM's graphics hardware tends to be conservatively engineered (slow) and conservatively priced (expensive), and it's up to the rest of the industry to push performance up and price down.

The technological foundation of the PC graphics industry rests with a few companies that have the resources to reverse-engineer a new IBM graphics standard and produce a chip set emulating its functions, right down to the register level. (Note that the 8514/A is becoming popular only as register-compatible chip sets become available.) These companies, the largest of which are Western Digital/Paradise, Headland/Video Seven, and Chips and Technologies, then use the chip set to make compatible products, sell the chips to other companies that make compatible products, or do both. Chips and Technologies almost never sells directly to end users; the others sell chip sets and boards to OEMs and completed boards to the public. Unfortunately, the fact that one chip set can be used by a variety of

board makers means it's unlikely that users know which chip set they're running, which can make software installation and troubleshooting very difficult.

Each PC graphics company has several things it can offer the public: chip sets made by the company itself; additional silicon (gate arrays) to add features to the base chip set; quick time to market; more or fewer software drivers, of greater or lesser quality; sales and distribution channels that move products quickly and efficiently (that is, cheaply) to market; and service and support. Different products have a different mix of benefits. This book is technically oriented and can't be revised often enough to keep up with each new board that appears, so we focus on chip sets, which we refer to by the names of the companies that produce them.

The Role of VESA

The VESA (Video Electronics Standards Association) organization was formed to create a monitor standard for 800 x 600 resolution, called super-VGA or extended VGA. Video-board manufacturers and monitor makers stand to profit from moving the mass market to higher and higher resolutions; however, the step from VGA to super-VGA had been messed up beyond belief, with a wild variety of incompatible boards, monitors, and software drivers. The original purpose of VESA was to define a resolution of 800 x 600 by 16 colors that would be the same from one board manufacturer to the next, so that all VESA-approved boards would support all 800 x 600 monitors and a single 800 x 600 VersaCAD driver, for example, would work on any VESA-approved board. This goal was achieved in the summer of 1989, promising improved hardware compatibility among video boards and monitors.

Several members of the VESA committee feared that their attempts to create a mass market for 8514/A-compatible boards would collapse

into the same morass of incompatibilities as the attempt to go beyond 640 x 480 by 16 with VGA. So the 8514/A subcommittee was formed.

The VESA organization includes all the leading PC graphics hardware companies, the largest of which are Western Digital/Paradise, Headland/Video Seven, Chips and Technologies, and ATI. Panacea has also been very active in VESA. The 8514/A subcommittee has been successful in establishing standards for passing VGA signals through an 8514/A board for PC compatibility and in defining standard register names across all VESA-approved boards. The latter effort is a great help in writing software and has been invaluable for putting together this book.

2

IBM's 8514/A

IBM's 8514/A Display Adapter was introduced, along with VGA, the PS/2 line of computers, the Micro Channel bus, and several monitors, in April 1987. The 8514/A had a pass-through connector for VGA, about twice as many pixels at 1,024 x 768 as VGA had at 640 x 480, and greater performance. Yet although VGA has rapidly moved to the leading position among video standards and the Micro Channel has had reasonably good, if not great success, the 8514/A has languished.

There are several reasons for this. VGA was built into the motherboards of high-end PS/2s, guaranteeing that it would be widely used. IBM provided detailed technical information about the VGA hardware, which made it easy to write software directly to the registers and easy to clone. First near compatibles and then true compatibles quickly appeared at rapidly decreasing prices. Some monitors being used for EGA display, such as the NEC Multisynch, had analog inputs and could already support VGA; others followed, and the price of both graphics cards and monitors quickly fell, first to less than $1,000 for each part and then to less than $1,000 for both together. The availability of VGA-specific software lagged at first, but this has changed.

By contrast, the 8514/A began life as a high-priced product ($1,290 for the 8514/A card with 512K of VRAM and $1,560 for the 8514 monitor) with little technical information. IBM not only did not provide a register specification for the 8514/A hardware, but the company also promised it never would. It also failed to provide much marketing or advertising support for the product or to lower its price over

time. Applications were supposed to be written to the Adapter Interface (AI), a high-level software interface of the same kind that the PC market had rejected again and again, to get the speed advantage of going directly to hardware. Video-board companies were also supposed to make AI compatibles rather than true register-level compatibles, and all of them decided not to do so, except for some TI 34010-based boards.

So software developers were stuck with the prospect of writing unnecessarily slow programs (because of AI overhead) to an unpopular graphics standard (sold only, at high prices, by IBM and available only for the Micro Channel bus). Many developers went ahead and developed drivers, but in some cases the effort was halfhearted, and because of the unavailability of technical information until now, almost all the drivers go to the AI rather than to the registers.

Despite the dismal prospects caused by the IBM-only, Micro Channel-only nature of the product, sales increased steadily over the first two years of the 8514/A's existence. By the time the first compatibles were introduced at the end of 1989, about 250,000 units had been sold. A small but growing number of drivers were written to the AI, including a driver from Autodesk for AutoCAD, or directly to the registers, including third-party drivers for AutoCAD such as Panacea's Display List Driver (DLD) 8514/A.

Ironically, IBM's AI-only strategy for the 8514/A was undermined by its software partner, Microsoft. Operating under what was rumored to be one of the strictest nondisclosure agreements in computing history, Microsoft was able to obtain a register specification for use in developing a hardware-specific driver for Windows and OS/2 Presentation Manager, which both IBM and Microsoft agreed had to run quickly on the 8514/A. This speeded the operation of programs running under Microsoft's graphical environments on the 8514/A but had an unfortunate side effect. Any third-party 8514/A board that was AI compatible but not register compatible would not be able to use the driver to

run OS/2 Presentation Manager, so making such a board was guaranteed to be a losing proposition.

The graphics-hardware companies applied their energies to the task of reverse-engineering the 8514/A. The first chink in the 8514/A's proprietary armor appeared in early 1989 with the publication of a register specification by Headland/Video Seven. This specification was obtained by many software developers, none of whom except Microsoft had been able to get a copy of IBM's official 8514/A specification. Western Digital/Paradise and Chips and Technologies soon followed with register specifications of their own. Because any company that had gone to the trouble to reverse-engineer the 8514/A registers could be assumed to be working on a compatible board, the race to market was on.

The primary reason IBM cited for trying to stick to the AI standard is that the company would later introduce more-powerful graphics cards that would be more powerful than the 8514/A and AI compatible but not 8514/A register compatible. (The company has done this with the Image Adapter/A, a 34010-like product that it has kept very quiet about.) So products written to the AI would be able to move up easily, and any register-compatible market would be a transient phenomenon. But by failing to offer the "next step" in more than two years, IBM gave its competitors plenty of time to decode the registers. And by giving the 8514/A hardware specification to Microsoft, the company signaled that a register-compatible product would indeed have a measurable life span.

As we are writing this, at the beginning of 1990, IBM has yet to make any announcements about its next graphics standard. There is now so much hardware and software momentum surrounding the 8514/A register standard, though, that any move IBM does make may be the foundation of the next graphics standard but is unlikely to have immediate mass-market impact.

Features of the 8514/A

For the feature set of the 8514/A to make sense, it must be remembered that it's an add-on card designed for systems with VGA on the motherboard. This requirement relieved it from the need to have BIOS code for start-up or software compatibility with existing programs — these were provided by the motherboard VGA. Unlike EGA and VGA, the 8514/A was almost completely free from the constraint that it directly support previous standards, allowing IBM to make a relatively simple and manufacturable board with good performance and advanced capabilities. (Three years later, with silicon capability having improved considerably, only one entry, from Integrated Information Technologies and Trident, tries to integrate VGA and 8514/A in a single chip set.)

The major features of the original IBM 8514/A Display Adapter, which we will explore in more detail later in this chapter, include the following:

- Dual-monitor support: any VGA-compatible screen from the motherboard and 8514, 8513, 8512, or 8507 support from the 8514/A add-in board.

- An on-board ROM with power-on self-test (POST) and initialization code but no BIOS.

- Two resolutions: 640 x 480 and 1,024 x 768 (in IBM's original 8514/A, the latter resolution is interlaced only).

- Built-in support for addressing a 1,024 x 1,024 drawing area. Because the maximum display is 1,024 x 768, at least one-fourth of the drawing area is available for off-screen storage.

- Two memory sizes: 512K base or 1 Mbyte with upgrade.

• Up to 256 colors at 640 x 480 resolution and up to 16 colors at 1,024 x 768 resolution, with 512K of memory. The memory upgrade to 1 Mbyte enables 256 colors at 1,024 x 768 resolution.

• A palette of 262,144 colors, regardless of memory size.

• Bit-block transfer (BitBLT) capability between system memory and the 8514/A and within a bit plane or between bit planes on the 8514/A. Supported in hardware.

• Advanced text modes supporting text sizes up to 255 x 255 pixels and text placement at any pixel address (via the AI).

• Rectangular clipping supported in hardware, greatly speeding windowing performance.

• Line-drawing and rectangle-fill functions supported in hardware.

• Hardware support for short-stroke vectors (SSVs), which are commonly used for fonts.

• Hardware support for raster operations (mixes) between two operands — Src (source) and Dst (destination); 16 logical and 14 arithmetic mixes.

• Object code for the AI, the only interface between software and the graphics card documented and supported by IBM.

• Text and graphics cursors supported by the AI.

• Line and area patterns supported by the AI.

• A tool kit with C bindings and examples.

• Three SSV fonts supplied with the board.

It's hard to exaggerate how important it is for an "8514/A compatible" to be completely compatible with each of the features in this list, because PC programmers will take advantage of every major and minor feature in order to get maximum performance. For instance, the Integrated Information/Trident 8514/A compatible puts 8514/A and VGA functionality into one chip and uses the same on-board RAM for both.

This approach is very efficient, but it overlooks the fact that the IBM specification has VGA and 8514/A separate, with separate memory areas. Programmers, including one of this book's authors (Richter), have kept separate images in the VGA and 8514/A memory areas and switched between them by transferring control from VGA to 8514/A and vice versa. This technique doesn't work with the integrated VGA-8514/A approach.

Also, although the 34010/34020 market is defined by a chip architecture, the 8514/A-compatible market is defined by the word *compatible*; if a high degree of compatibility isn't maintained, the market quickly shrinks. Programmers and users can't be expected to maintain slightly different versions of programs, depending on which chip set is on the board they're using. The VESA 8514/A subcommittee is supposed to be creating and maintaining a compatibility standard for 8514/A , but this type of standardization takes time, as there are many delicate areas to discuss and decide upon. So the only measure users can count on for now is IBM 8514/A compatibility, and the only thing programmers can do is write to a least common denominator that includes as many boards as possible.

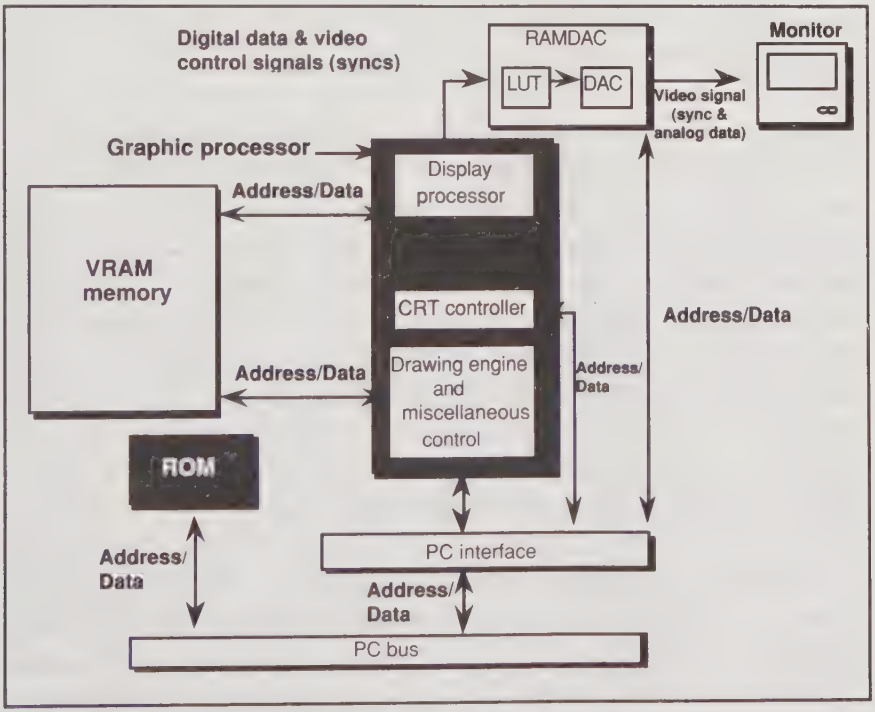
Architecture of the IBM 8514/A

The IBM 8514/A's design is fairly simple. Two chips — a drawing engine and a display processor — make up the heart of the system. The drawing engine takes commands from the CPU, decodes them, and executes them. Executing the commands usually involves updating

video memory, a lookup table (LUT) for color values, or other control registers. Commanding the drawing engine, updating the contents of video memory, and updating the contents of the control registers are the main concerns of programmers.

A lot of functions are included under the heading of drawing commands. Line drawing, bit-block transfers (BitBLTs), and combinational functions called mixes (to superimpose one object upon another, for instance) are all important.

Figure 2-1: Block diagram of the 8514/A



The display processor configures itself using the control registers, reads video memory, and outputs an image to the screen. It also con-

verts signals from an on-board clock to synchronization signals needed by the monitor. The display processor reads a pixel value and uses it as an index into the LUT to get the red, green, and blue (RGB) values to display. A separate digital-to-analog converter (DAC) for each primary color then converts the three digital values to three analog ones for display on the attached analog monitor. The LUT and DACs are most often combined on one chip, called a RAMDAC.

Color Math

The LUT is 18 bits — 6 bits each for red, green, and blue. This means it can have 2^6 (64) shades each of red, green, and blue and can display 2^{18} (64^3), or 262,144, possible colors. Note that the size of the color palette is dependent on the size of the LUT and not on the amount of memory in the video subsystem.

The number of colors that can be displayed at any one time is dependent on the amount of memory. A resolution of 1,024 x 768 means there are 786,432 pixels. At 4 bits per pixel (16 colors), 384K is needed to store the image, so half a megabyte of video memory is required; at 8 bits per pixel (256 colors), 768K are needed, so 1 Mbyte of video memory is required. This means that the 8514/A can display either 16 or 256 colors out of a palette of 262,144 colors, depending on whether half or a full Mbyte of memory is available.

Implementing the Architecture

The IBM 8514/A implements the drawing engine and the display processor as separate chips that comprise the graphics processor, with a separate RAMDAC, but this exact chip configuration isn't required. Different compatible boards can implement the same functionality in different types or numbers of chips without affecting the register-level programming interface, let alone the AI level. Looking at the board as a black box, it's plain that in addition to the processor, there must be some way to get data and commands in and to get the

displayed image out. This is the concern of other functions on the 8514/A board, external to the graphics processor.

One important function is the PC interface, which handles data transfer between the PC and the drawing engine (for commands) and video memory (for data). Communications between the PC and the 8514/A are handled by reading from or writing to I/O ports on the PC. Data can be transferred back and forth from video memory to the PC, but commands can go only one way. Direct memory access (DMA) is not implemented on the 8514/A, although it is on some compatibles.

The IBM 8514/A has either 512K of memory (4 bits/pixel or 16 simultaneous colors) or 1 Mbyte of memory (8 bits/pixel or 256 simultaneous colors). The palette size is always 262,144 colors. The original 8514/A always uses VRAM (video RAM), not the cheaper DRAM. (We use the terms *video memory* and *display memory* to mean whichever kind of memory is in use on a given video card, because some boards use only VRAM, some use only DRAM, and some use either.)

Although DRAM can be almost as fast as VRAM and costs considerably less, it's single-ported: It can't be read at the same time as it's being written. Every time the screen is redisplayed — 43 times per second for IBM, 60 or even 70-plus times per second for noninterlaced compatibles — video memory is read. Because video memory must be read a fixed number of times per second, writes to a given location must wait until that location is not being read, which can introduce several wait states and so slow screen updating. VRAM can be simultaneously read and written, so video memory can be updated without wait states and without interrupting the screen display. IBM chose to go with VRAM, despite the expense.

The Processor and Registers

Communication with the drawing engine occurs through a set of registers, listed in Table 2-1. The processor acts on changes to the contents

of the registers; for instance, a write to the Current Y Position register causes the cursor to move to the new *y* position. However, a sometimes confusing set of setup commands must be issued before actually "doing something"when writing directly to the registers, and mistakes are easy to make.

Table 2-1: 8514/A registers, grouped by function

Port Address (hexidecimal)	Access	Description
Lookup Table Registers		
02EA	R/W	DAC Data
02EB	W	DAC Read Index
02EC	W	DAC Write Index
02ED	R/W	Palette Data
CRT Control Registers		
02E8	R	Display Status
02E8	W	Horizontal Total
06E8	W	Horizontal Displayed
0AE8	W	Horizontal Sync Start
0EE8	W	Horizontal Sync Width
12E8	W	Vertical Total
16E8	W	Vertical Displayed
1AE8	W	Vertical Sync Start
1EE8	W	Vertical Sync Width
22E8	W	Display Control
4AE8	W	Advanced Function Control
Miscellaneous Control Registers		
42E8	R	Subsystem Status
42E8	W	Subsystem Control
46E8	W	EPROM Page Select
Drawing Control Registers		
82E8	R/W	Current Y Position
86E8	R/W	Current X Position
8AE8	W	Destination Axial Step Constant
8EE8	W	Destination Diagonal Step Constant
92E8	R/W	Error term
96E8	W	Major Axis Pixel Count

9AE8	R	Graphics Processor Status
9AE8	W	Drawing Command
9EE8	W	Short Stroke Vector Transfer
A2E8	W	Background Color
A6E8	W	Foreground Color
AAE8	W	Write Mask
AEE8	W	Read Mask
B2E8	W	Color Compare
B6E8	W	Background Mix
BAE8	W	Foreground Mix
BEE8	W	Multifunctional Control
Index 0	W	Minor Axis Pixel Count
Index 1	W	Top Scissors
Index 2	W	Left Scissors
Index 3	W	Bottom Scissors
Index 4	W	Right Scissors
Index 5	W	Memory Control
Index 8	W	Fixed Pattern — Low
Index 9	W	Fixed Pattern — High
Index A	W	Pixel Control
E2E8	R/W	Pixel Data Transfer

Some of the registers, such as the Destination Y Position/Axial Step Constant drawing-control register, are multipurpose. The Multifunction Control register, as its name implies, has several purposes, depending on the value of a sequence of bits in the data being written to it — another case in which programming is a bit tricky and in which obscure errors can have serious consequences.

The CRT-control registers determine the resolution and frequency of the screen display and are relatively straightforward to program. Either 1,024 x 768 or 640 x 480 resolution is supported; the IBM 8514/A outputs an interlaced signal at the higher resolution, causing noticeable if not terrible flickering on-screen.

The miscellaneous control registers vary in usefulness. The ROM Page Select register is used to get at information in the 8514/A's ROM, which programs do not normally need to access directly. The Subsystem Control register, though, controls interrupts, an important function. It can be used to request that an interrupt be issued on verti-

cal synchronization (at which point animation commands, for instance, can be issued), on filling or emptying of the command queue, or on a data-ready signal from the graphics processor. Reading the Subsystem Status register (which sits at the same I/O address as the Subsystem Control register) allows you to check the status of the interrupts; writing to it allows you to clear them.

The drawing-control registers support many different functions, but several of them are usually manipulated only during initialization and are then left unchanged. The Graphics Processor Status register shows the current state of the command queue, which can hold up to eight commands simultaneously. Commands in the queue can include foreground and background mixes, line-drawing parameters, *x* and *y* start positions, and actual drawing requests. Applications should always check the state of the queue before issuing drawing commands to make sure enough space is available to hold them.

The Command register is used to request a graphics operation such as line drawing (which needs the results of calculations for line angle and so on in order to proceed), rectangle fill, polygon fill (tedious to set up, fast to execute), and short-stroke-vector drawing. Flags in this register specify when reads should be performed instead of writes, when PC data should be used instead of color register information as a data source, when bytes should be swapped, and when 16-bit rather than 8-bit I/O should be used.

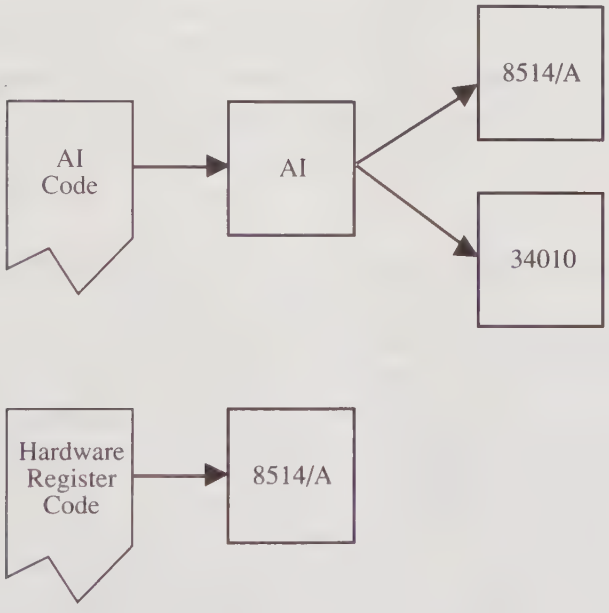
The Pixel Data Transfer register is used for all data transfer, such as loading an image from PC memory into video memory. This is simple for programming, but it's a real bottleneck, especially for frequent updates of large complex images or for saving a screen image before a task switch.

The registers, most of which are 16 bits wide, are accessed via I/O addresses. The I/O register addresses mostly have the lowest 10 bits in common and differ in the upper 6, so that up to $2^6 = 64$ registers can

be differentiated, more than enough for the 36 registers on the 8514/A.

The AI takes commands from a program and first checks all relevant parameters and then issues commands to (or reads information from) the 8514/A registers. Programs that write to the registers directly skip the AI and read and write the registers themselves.

Figure 2-2: Program writing to the AI vs. writing directly to the registers



Mixes

The 8514/A has a wide range of raster operations and combinatorial functions, which IBM calls mixes. These allow the final value written to a pixel to depend on logical operations applied to a source pixel and the destination pixel. The source pixel can come from the color registers, PC memory, or 8514/A memory; the destination pixel is a specific pixel already in 8514/A memory.

A simple example would be inverting a block of pixels to highlight them by using the NOT operator to reverse their contents. Another example is the most commonly used mix, Source XOR Destination, used for cursors and rubber-band lines, because applying the same mix again causes a return to the original destination value.

Most graphics boards implement transparency by treating the color 0 as a null or transparent color. The 8514/A has a much more sophisticated approach, using "color comparisons" in which the destination is compared to a value in the Color Compare register. A drawing operation is then either performed or not, based on the result of the comparison. This approach allows certain colors or ranges of colors to be protected from updating or to be always updated for specific operations. Interesting effects can be obtained from this "write protection" of specific colors.

Memory Architecture

There are two types of memory architecture for video boards: planar and packed pixel. A packed-pixel architecture, as is used on most high-end graphics boards, treats several bits in a row as a pixel. In 4-bit-per-pixel mode, 2 pixels would fit in a byte; in 8-bit-per-pixel mode, only 1 pixel would fit in a byte.

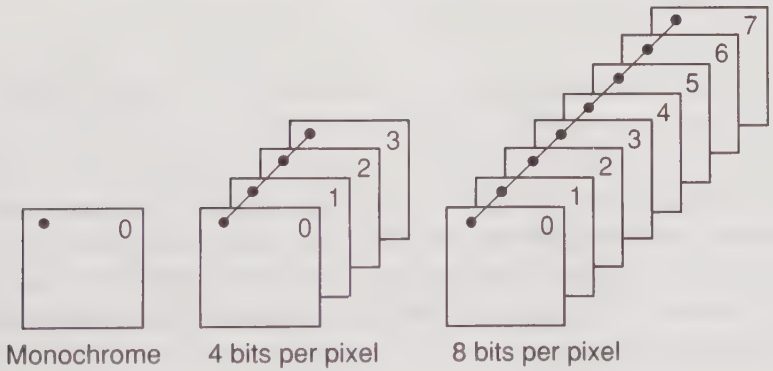
Planar architecture, as used by EGA, VGA, and the 8514/A, divides memory into planes, 1 pixel thick, stacked on top of each other. So the first block of memory gives the contents of the first bit plane, the second block gives the contents of the bit plane on top of that, and so on. The final color of a given pixel depends on the value stored in all the bit planes then in use, which can be anywhere from 1 (monochrome display) to 8 (256 colors). This allows pixels to be drawn at the same speed whether 1 bit per pixel is used or 8.

The 8514/A is very flexible and auto-converts from packed-pixel storage formats to its own planar architecture. This means that whereas

EGA and VGA forced programmers to work in planes, the 8514/A allows them to work with packed pixels and makes the conversion to planes internally.

The planar architecture also explains why the 8514/A supports the same number of pixels, 768,000 worth, regardless of the amount of memory it has, but supports only 4 bits per pixel (bpp) in the base configuration and 8 bpp with memory expansion. The memory expansion just adds four more planes on top of the original four, rather than adding memory and additional pixels after the original memory. This suggests that more bit planes could be added with additional memory beyond 1 Mbyte plus some minor control hardware. Thus, 2 Mbytes of VRAM would support 16 planes, and 3 Mbytes would support 24 — what's currently called true color on Macintosh and PC video boards. It's worth noting that the 8514/A AI supports pixel sizes up to 32 bits.

Figure 2-3: The final displayed color of a pixel is the result of overlaying 1, 4, or 8 separate bitplanes



Font Support

The IBM 8514/A supports several helpful features for handling fonts. However, the board does not have a text mode per se; all characters

are treated as graphics. The AI does have specific support for three font sizes, which are usually used as described below. The following font-related features come with the 8514/A and the AI:

- Support for short-stroke-vector (SSV) and bit-mapped fonts in a variety of sizes.

- Three SSV fonts supplied with the board.

- Advanced text modes supporting text sizes up to 255 x 255 pixels and text placement at any pixel address.

- Up to four alphanumeric fonts addressable simultaneously. Three SSV fonts are supplied with the board, and IBM's AI programmer's tool kit comes with about a dozen more fonts. User-defined bit-mapped and SSV fonts are also supported.

- Three character sizes at 1,024 x 768 resolution: 12 x 20 pixels (38 rows of 85 characters), 8 x 14 pixels (54 rows of 128 characters), and 7 x 15 pixels (51 rows of 146 characters).

- One character size at 640 x 480 resolution: 8 x 14 pixels (34 rows of 80 characters).

The MDA took ASCII codes from the CPU, mapped them onto a set of characters stored in ROM, and displayed the ASCII characters on-screen. The 9 x 14-pixel character block and some clever font design resulted in crisp, clear characters that haven't been improved upon by any IBM standard until the VGA and 8514/A.

However, user requirements have increased since the introduction of the MDA ten years ago. Users now want to display a variety of fonts on-screen to match the high-quality laser printer output to which most people now have access. This is usually done by storing bit maps for each font, in each size the user might wish to use, on-disk. Bit

maps for a single font in a moderate variety of sizes require about 70K.

IBM has gone a good step away from this problem with the 8514/A by defining characters as a set of lines or as a bit map. The name *short-stroke vector* refers to the short lines with which the fonts are drawn. Three SSV fonts are included on a disk with IBM's 8514/A board.

The 8514/A and VGA

The 8514/A differs from previous IBM graphics standards, each of which included compatibility with previous standards. The only exception is the PGC (Professional Graphics Controller), an 8088-based board that was introduced in 1984 and that offers 640 x 480 (256 colors) resolution and a CGA-compatibility mode. This board met a narrow market need, was never reverse-engineered by the *major* graphics companies, and was never a big success in the marketplace.

At introduction, the 8514/A has avoided this fate for several reasons. It offers a quantum leap over existing resolution standards and has more processing power to get screen images up quickly. And it was introduced only for the high end of the PS/2 line, which uses the Micro Channel bus (the only bus with which IBM's 8514/A works), and has VGA on the motherboard. VGA is passed through the 8514/A, providing backward compatibility with existing software (MDA, CGA, and EGA) and the ability to start itself up under DOS. On the PS/2's Micro Channel motherboard, one of the slots has an additional connector for VGA video pass-through. The 8514/A is plugged into this connector, and a software switch tells the 8514/A to let the VGA signal pass through to the screen or supersede the VGA signal with its own output.

There is a problem here, however. The signal sent through to the 8514/A has not yet passed through the VGA LUT, so only pointers to

colors are included, not the actual colors. If these pointers were sent through the 8514/A's LUT, the results would be undefined — that is, anything but what you wanted. To fix this, any write to the VGA palettes while the 8514/A is displaying the VGA signal automatically updates the 8514/A's LUT as well, so the result is the same as if the VGA's LUT were used. There are two other important points here: Unlike the VGA LUT, the 8514/A LUT can be read and therefore saved; and the 8514/A LUT can be written to directly, without affecting the VGA LUT.

The problem is that all this forces a system to have VGA capability in addition to 8514/A. To be useful as a system's sole graphics adapter, a graphics board needs two things:

- A BIOS (basic I/O system) to allow it to initialize itself and put text, at least, up on the screen.

- Compatibility with enough software that users can get their work done.

The 8514/A has neither of these. In place of a BIOS, the IBM 8514/A has a small amount of ROM for use by the PS/2's power-on self-test (POST) initialization procedure. A BIOS would have added a small amount to the board's cost but not provided a difficult technical obstacle. The real barrier is compatibility with a lot of software. IBM had the choice of providing 8514/A drivers for most or all DOS programs with the board on the day it shipped; including VGA (and therefore EGA, CGA and MDA) capability on-board; or providing the pass-through mechanism and requiring that VGA be present as well.

Writing drivers for most or all DOS programs in a reasonable amount of time was impossible. Including VGA with 8514/A, at the time 8514/A was introduced, would either have limited the capabilities the 8514/A could offer or have been very expensive if not impossible. So IBM chose the last option — providing a pass-through and requiring VGA. Although this was an easy choice for PS/2 systems, with

their motherboard VGA, the VGA pass-through requirement and IBM's failure to offer an AT-bus board have until now greatly limited the 8514/A's popularity.

Functional Highlights

The original IBM 8514/A with the AI was a Good Thing. It showed how to implement a relatively inexpensive near-megapixel display with up to 256 colors from a large palette, and it established a standard for others to emulate. Actually, it established two standards: the Adapter Interface, which IBM wanted everyone to use; and the register standard, to which most of the major PC graphics companies are making compatibles.

The advantages of the IBM 8514/A were numerous. The most important were:

• Cheapness. The original board with 512K of video RAM, a rare and expensive commodity at introduction, cost $1,290, and the 8514 monitor cost $1,560. Although neither product was less than the $1,000 mark, which is thought necessary to start creating a mass market, for a high-end board with Big Blue's name on it, the 8514/A was a good deal.

This cheapness was achieved at some cost, mainly the use of interlacing at 1,024 x 768 resolution. But it was a sensible compromise at the time and one that has been superseded by compatibles now that the necessary components are less expensive.

• The Adapter Interface. Although programmers typically deride software interfaces and seek any means to get past them (part of the purpose of this book), the AI was not too inefficient (about twice as slow as hardware programming) and set a standard that other products such as TI-based boards and future IBM products could match

without duplicating (and accepting the limits of) IBM's hardware implementation.

• Fast line drawing. The 8514/A was introduced at the same time as VGA, and drawing lines faster on a 1,024 x 768 screen than on a 640 x 480 screen, even after AI overhead, was quite an accomplishment. The even higher speeds achievable by writing to the hardware and/or using faster compatibles extend the useful life of the product.

• Fast data copies. Considering that all data must go through the Pixel Data Transfer register, the 8514/A moved data pretty quickly. Again, enhanced compatibles, especially those with DMA (direct memory access) capability, improve performance further.

• Expandability. The 8514/A had a built-in upgrade path from 512K (4 bits per pixel, 16 colors) to 1 Mbyte (8 bits per pixel, 256 colors), with not-yet-fulfilled hints of further expansion.

• Set a standard. The rest of the PC graphics industry has not yet shown itself capable of setting a hardware standard. TI is well on the way with the TI 34010/34020, the BGA chip, and TIGA. The major PC graphics companies, grouped as VESA, have finally established a super-VGA standard at 800 x 600 but have not yet been able to agree on a 1,280 x 1,024 super-8514/A standard, although it's probably needed. IBM is, at least for now, still the primary PC-graphics-standard setter.

Despite all the advantages listed here, the 8514/A has been slow to catch on. Only now, with hardware compatibles becoming available, is the IBM version becoming popular, with about 250,000 units (Dataquest projection) sold through the end of 1989. This compares poorly to the millions of EGA and VGA boards sold. The main problem has been the lack of a compatible market to make the boards widely available and cheap. Other problems, many of which are solved by some or all of the 8514/A compatibles, are described in the next section.

Functional Limitations

Problems that have slowed the acceptance of the IBM 8514/A include:

• Micro Channel only. The 8514/A was available only from IBM and available only for the Micro Channel bus. There was no way for AT-bus users to get it or for a company with a mix of Micro Channel- and AT-bus PCs to standardize on it to any significant degree.

• No hardware compatibles. Even if everyone politely agrees to use the AI, hardware compatibles are necessary to take the standard out of IBM's sole control and into the hands of the marketplace, at which point users trust that it will be around for a while.

• Unreadable registers. There are two ways for a software program — whether it be the main program being run by the user, a memory-resident program, or a multitasking control program — to keep track of graphics status. The easy way is to examine the current register status — but most of the 8514/A's registers are unreadable (see Table 2-1). The other is to create a data structure like the 8514/A register set in memory and then update it through all relevant graphics commands, which is hard to do reliably and slows performance. This makes it hard for several kinds of programs to run on an 8514/A-based system.

• Unnecessary complexity in using registers. Complicated special calculations and register setup are needed to do simple things such as drawing a line; this is unnecessary, mistake-prone, and cumbersome.

• No DMA. All memory transfers are through a single Pixel Data Transfer register, so operations such as loading and saving a screen image (up to 768K) are slow.

• Not a stand-alone board. The 8514/A works only if VGA (or another IBM standard display board) is already present — an extra expense.

• "Only" 8 planes of color. A 1-Mbyte 8514/A has more than enough memory to support 16 color planes at 640 x 480 resolution, but the planar architecture of the 8514/A limits the number of color planes, regardless of resolution. There's also no support for additional memory to give additional color planes at any resolution.

• Interlaced display at 1,024 x 768. This is a big limitation — many CAD users can't work with the kind of flicker that thin horizontal lines produce on an interlaced screen. IBM has never defined 8514/A support for noninterlaced screens at 1,024 x 768.

• No hardware support for graphics primitives such as circles, ellipses, arcs, or polygon fill. This slows both execution and programming.

• No 1,280 x 1,024 resolution. This is a wonderful resolution, partly because a 1,280 x 1,024 screen can show four full 640 x 480 windows at once. It's possible to push a 1,024 x 768 monitor design to 1,280 x 1,024 — but not much further — without greatly increasing cost. It would be a real advantage to have a standard way to go to this next logical step within the 8514/A architecture, but unfortunately there isn't one. This leaves the field to the TI 340x0 and a few other competitors.

Most of these problems have been addressed by at least one, if not all, of the compatible makers. Most of the missing capabilities can be provided without sacrificing AI or register compatibility, although there is no compatible way to access 1,280 x 1,024 where it's available. In the next chapter, we'll give an overview of the tasks facing compatible makers and developers of software designed to run on the AI and/or the registers as an introduction to our programming and board-specific chapters to follow.

3

Software Development for the 8514/A

The simultaneous arrival in the mass market (less than $1,000 for a board) of the 8514/A and 34010 boards running TIGA, with boards having different amounts of video memory and different extensions to standard modes, makes decisions about software development difficult. Even after you've decided that it is indeed worth writing an 8514/A driver of some type, you have to makes several trade-offs. Each of the considerations that affect the type of driver or drivers you'll write is listed and discussed in this chapter. The most important single choice may be whether to write to the 8514/A Adapter Interface (AI) or to the registers; we take up that issue in several of the following sections.

Standards

The whole reason for the mass market in PC compatibles is the existence of standards, most of which have been defined by IBM. Much of the popularity of standards derives from the fact that they make possible mass production, leading to lower prices. Because of the conflicting drives toward lower cost and higher functionality, there are four different hardware configurations that users refer to as "the 8514/A":

@ 8514/A register compatible with 256K of memory. This configuration is not part of the original 8514/A standard; it's an approach being tried to give an easy upgrade path from a fast, cheap system to a

full-blown 512K or 1-Mbyte 8514/A system.The 256K configuration gives fewer choices as to resolution and number of colors (up to 640 x 480 by 16 colors or 1,024 x 768 by 4 colors), and performance is hurt by the fact that DRAM rather than VRAM is used for video memory. This configuration may be built in to many PC motherboards, and monitors that support 640 x 480 by 16 colors will be cheap. These same systems will also support VGA drivers, using the same display memory as would be used by the 8514/A, so programmers can't count on using VGA and 8514/A display memory for different screen images.

@ 8514/A register compatible with 512K of memory. This is what will be most widely meant by "8514/A compatible," as it's the minimum configuration for most add-in boards. It allows resolutions up to 640 x 480 by 256 colors or 1,024 x 768 by 16 colors.

@ 8514/A compatible with 1 Mbyte of memory. This amount of memory allows 640 x 480 by 256 colors, with room to spare, and up to 1,024 x 768 by 256 colors.

@ Any hardware that runs the 8514/A AI. This includes all the configurations listed above, plus many TI-based boards and the IBM Image Adapter/A, which uses an extended version of the AI and is not hardware compatible with the 8514/A registers.

There will also be 8514/A compatibles that support 1,280 x 1,024 resolution, but for now, each chip set will support this resolution in a different way. Although we note which chip sets have extended modes in this book, we expect that only very demanding applications will have drivers that take advantage of these incompatible extensions.

Performance

The second most important factor driving the PC market is performance within a standard — and often the standard is bent in pursuit of performance. This is certainly true in the case of the 8514/A, be-

cause IBM tried to set the AI and not the hardware as the standard, only to have first Microsoft, with Windows and OS/2 Presentation Manager, and then PC graphics companies go straight for a hardware standard.

With the IBM 8514/A board, there's about a two times performance difference between going to the hardware and writing to the AI. This may not be true of compatible boards, though. Compatible makers claim to have improved their hardware and their versions of the AI so that overall performance is better for a given driver — about double IBM's is the most common claim. Intriguingly, they also claim that they've eliminated much of the performance overhead in their AIs, so that the penalty associated with it is much less than two times. However, programmers must take into account that the IBM 8514/A board is fairly popular, so its performance must be acceptable or users of the IBM board won't like the product. In some cases, writing to the registers will make the difference between acceptable and unacceptable performance, especially with the IBM board.

AI code can basically be considered an "interpreted" language, which is decoded by the board every time it's run, whereas register-level code can be considered "compiled" and ready to go directly to the hardware. It's unfortunate that there's not, at this writing, an "AI compiler" that can convert AI code to register-level code that runs on all hardware compatibles without programmers having to learn the gory details; with such a product, it would be easy for developers to offer drivers for both the AI and the registers.

Besides the choice of writing to the AI or the registers, several other factors affect the 8514/A's performance. The resolution used certainly affects perceived performance, because the time to update the screen depends on how many pixels have to be driven to it. However, the number of colors used does not directly affect performance. The 8514/A is very efficient about translating bit patterns to colors and using the LUT to get the correct colors on the screen.

There is overhead involved with using color, though. Setting up and changing colors takes time, and when data is moved to and from video memory or from one location to another within memory, the use of more bit planes multiplies the amount of data that must be transferred. However, the 8514/A's pseudoplanar architecture minimizes the performance hit. Color does not affect performance as dramatically as resolution does.

Text Display

Many programs are most concerned with getting a lot of text on the screen with as high a legibility as possible. The 8514/A AI has four default text modes, any of which can be used on any 8514/A-compatible board.

The predefined text modes are shown in Table 3-1. They range from 80 characters by 34 lines, at 640 x 480 resolution, up to 146 characters by 52 lines, at 1,024 x 768 resolution. Choosing a text made can be a tough decision because text that looks just right on a smaller screen (14 inches or so) looks cartoonishly large on a bigger one (19-inch screens are now being used with PCs). In some cases, it may be possible to make the text size a user-selectable option.

Table 3-1: 8514/A default AI text modes

Screen Size	Alpha Text Size	Text Resolution
1,024 x 768	12 by 20	85 by 38
640 x 480	8 by 14	80 by 34
1,024 x 768	8 by 14	128 by 54
1,024 x 768	7 by 15	146 by 52

Performance is also affected by the choice of text size; scrolling and screen updates are much faster when there are fewer characters on the

screen. A smaller number of characters on-screen yields a double performance benefit: Fewer bytes have to be moved around, and the expansion of the smaller number of characters from ASCII text to bit maps or short-stroke-vector (SSV) fonts is much quicker for fewer characters.

The AI may be sufficiently fast for many text-based operations, especially at 640 x 480 resolution. This gives programs the maximum amount of portability and sufficient performance; a register-level driver may be used for the same choices or resolutions or only for higher resolutions.

Compatibles and VGA Support

Now that compatible boards are appearing, several approaches to the problem of VGA compatibility are being tried. The easiest choice is simply to keep the pass-through requirement; the 8514/A board then doesn't require its own BIOS or backward compatibility, and the board can be sold profitably to the huge installed base of 8514/A users. Western Digital even offers a board with a complete VGA, with 256K of memory, and complete 8514/A, with 512K or 1 Mbyte, all on one board.

The other approach is to make an 8514/A chip set that has built-in support for VGA. Until recently, most chip makers thought this was impossible. However, Integrated Information Technologies (a chip company) and Trident (a graphics company) have combined forces to create a RISC-type chip set with both 8514/A and VGA support. DRAM- and VRAM-capable chip sets are being offered. The chip set runs as a VGA on start-up, and it always stays a VGA if only 256K of memory is available. But if more memory is used (a total of 512K or 1 Mbyte), the board can also have 8514/A capability.

This is a technical triumph and makes it possible for system makers to integrate VGA graphics easily on the motherboard, with 8514/A

available simply by adding a small amount of RAM. There is a problem, however. Systems that use separate VGA and 8514/A implementations have two separate memory spaces — at least 256K for VGA and at least 512K for 8514/A. A small number of advanced drivers put one screen image in the VGA buffer and another in the 8514/A buffer and then quickly switch between the images by switching from VGA to 8514/A. By using a single block of memory for both, the combined solution is incompatible with these advanced drivers. This problem will have to be addressed by new versions of the drivers and will mean that the combined solution will always be slightly less compatible than boards that offer separate support.

Portability

Users and programmers will definitely be concerned with the number of PCs on which a given driver will run. The conflict between wanting to go to the least common denominator and getting maximum functionality is really in evidence here. Any 8514/A driver, AI or register-based, is an improvement over a comparable VGA driver because of the power of the 8514/A drawing engine. And the relatively high level of standardization that we can hope for from 8514/A hardware compatibles makes writing a driver more worthwhile.

Despite what promises to be wide popularity of 8514/A hardware compatibles, though, there are a lot of reasons to write to the AI. TI-based boards may be found most often among the most performance-oriented users, who are good people to reach with a software product. The same goes for upcoming IBM products that will also support the AI.

Portability concerns also affect what resolutions programmers want to support. Table 1-1 shows what resolutions and number of colors are supported by a given amount of video memory; programmers will have to consider the purposes of a given program and get a rough idea of the "hardware profile" of their target audience before deciding what

resolutions to include. Some users with 8514/A boards will have as little as 256K of video memory, which may be DRAM or the faster VRAM, and a monitor that supports only 640 x 480 by 16 colors, whereas others will have 1 Mbyte of fast VRAM and a 19-inch, 1,280 x 1,024 analog screen. Informal or even formal market research may be needed to make a good decision.

Ease of Development and Maintenance

Given the number of different platforms on which a "PC compatible" program, especially a graphically oriented one, is expected to run, ease of development and maintenance are now being considered much more seriously than traditional concerns such as raw performance. IBM tried to address this issue by defining the 8514/A in terms of the AI and not the hardware registers.

This attempt was undermined somewhat by the relatively slow performance of the original IBM 8514/A and because IBM hasn't chosen to offer an improved hardware- and software-compatible version of its own, so anyone who wants to "buy Blue" gets a relatively slow board, with an interlaced display of 1,024 x 768 to boot. But the improved performance of hardware compatibles and the claimed reduction of AI overhead means that an AI-based driver will be very likely to have acceptable performance — except, ironically, on IBM's 8514/A board and on a few TI-based boards that may run the AI more slowly than 8514/A hardware compatibles.

Writing to the AI is still the best choice for ease of development and maintenance, however. The overhead of the AI consists largely of error detection and prevention, which make software development easier, and AI code is easier for programmers to read and understand than is code written to the registers.

Being more understandable, code written to the AI is also easier to convert to TIGA and other graphics standards, which is important not

only because of the increasing variety of PC platforms but also because many PC programs end up being ported to other computers. Graphics may be the single most difficult item to translate for programs that depend on high-resolution displays.

A Development Strategy

One development strategy that many shops may choose to follow is to develop and test their drivers for the AI first. This is the quickest way to complete a driver that will work with the maximum possible number of boards, and programmers can get support from the widest possible range of sources. Then performance testing, user acceptance, and market realities can determine whether the driver should be rewritten to go directly to the hardware. It may even be worthwhile, whether through AI extensions or register extensions, to use certain chip-set-specific features, such as 1,280 x 1,024 resolution.

Programmers may even wish to automate the conversion process somewhat. It wouldn't be that difficult for someone who has done a few such rewrites to write a program to convert AI code to register code. Such a program may even become available commercially.

The programming chapters that follow are written to support this strategy. Although Chapters 4 and 5 together cover the AI and Chapters 6 and 7 together cover programming to the registers, anyone seeking to get the most out of the 8514/A should read all four chapters. The initial chapter in each pair (Chapters 4 and 6) are reference-oriented chapters that you should at least skim for broad familiarity with overall capabilities. The subsequent chapter in each case has programming examples of how to use the AI and registers, respectively.

Writing AI programs is a good way to learn the capabilities of the 8514/A without getting bogged down in debugging efforts, even for programmers who intend to work largely with the hardware regis-

ters. And familiarity with the register level helps AI programmers know what's going on under the covers. Even programmers who intend to work largely through Windows or OS/2 Presentation Manager or interested laymen can benefit from learning about both AI and register capabilities.

4

Programming the AI

This chapter is a reference for using the IBM Adapter Interface (AI) under MS-DOS.

As you have seen in the previous chapters, the AI is a reasonably well rounded software interface. Although there is a definite speed advantage in going to the 8514/A registers, the AI does provide portability across non-8514/A-compatible hardware. Many TI 34010 graphics-board manufacturers have shipped or are shipping an AI for their boards, and IBM's new graphics board is reported to be AI compatible.

Besides its portability to other platforms, including whatever IBM will come out with, the AI has another benefit: It's cheap. Getting documentation for the AI requires a call to the IBM Technical Books ordering department ([800] IBM-PCTB) and a major credit card. The base technical reference should cost you about $10 (P/N 68X2248, *IBM Personal System/2 Display Adapter 8514/A Technical Reference*), and a programmer's guide is also available for approximately $27 (P/N 68X2279, *IBM Personal System/2 Display Adapter 8514/A Adapter Interface Application Developer's Guide*). The technical reference has several typos and omissions, but it still provides the basic information necessary to understand the AI. The programmer's guide is really handy because it comes with a disk containing C language bindings and miscellaneous utility programs that make using the 8514/A easier to understand. The C language bindings and a selection of fonts are also shipped with each IBM 8514/A board.

Accessing the AI

From a programming perspective, the AI is modeled somewhat after existing graphical interfaces you may be familiar with. The AI is loaded as a terminate-and-stay-resident (TSR) program, and you must make sure you load the correct version for your board: 4 bits per pixel or 8 bits per pixel. If you run the wrong version, you may end up with some odd visual effects on your display. With the IBM version of the AI, the TSR program is called HDILOAD.EXE. The AI installation program detects the board configuration and copies the correct version of the AI into HDILOAD.EXE. There are no command-line options available for HDILOAD.EXE. Once you have loaded the AI, you can run an AI application. An application that uses the AI usually performs these steps:

1. Verifies that the AI is loaded.
2. If the AI is installed, "opens" it for use.
3. Initializes the graphics state of the AI and board.
4. Performs all the drawing and other graphical operations necessary for the application.
5. "Closes" the AI.

Because the AI is a TSR program, applications need some way to communicate with it so that it can issue the appropriate commands to the graphics board. When the AI is installed, it grabs interrupt vector 7FH. The various AI functions are actually invoked via a jump table located in the AI TSR program. The address of this table can be obtained via the following assembly-language code fragment:

```
;
; Check to see if the AI is loaded by checking the address
; at interrupt vector 7FH. If the address is 0:0, AI is not
; loaded.
;
        xor     AX, AX
        mov     ES, AX
        mov     AX, ES:[7FH * 4]
        or      AX, ES:[7FH * 4 + 2]
        jz      AI_Not_Installed
```

```
;
; AI might be loaded, call whatever function is at interrupt
; 7FH with a request for the jump-table address. This
; requires calling INT 7FH with a value of 0105H in AX.
;
        mov     AX, 0105H
        int     7FH
        jc      AI_Not_Installed
;
; If we make it this far, the pointer to the jump table is
; returned in CX:DX.
;
```

If you use the C binding, the routine getafi() does this for you:

```
/*
   Sample C code to determine if AI is loaded, and if so,
   get the pointer to the jump table. The jump-table pointer
   is stored internal to the C binding.
*/

   if (!getafi())
      {
      printf("AI not loaded!\n");
      exit(1);
      }
```

(Note: The source code for getafi() and other C-callable AI functions is in CALLAFI.ASM, which comes with the IBM 8514/A or the AI programmer's guide referred to earlier. Also necessary are the include file AFIDATA.H and IBMAFI.H, which contain definitions necessary to use the AI with C.)

Once the jump-table pointer has been obtained, calls can be made to the AI. To pass a command to the AI, you need to push the 32-bit segment:offset address of an AI function parameter block onto the stack and make a far call to the appropriate entry in the jump table.

Function Parameter Block

The AI function parameter block is a data structure whose actual contents and form vary among the different AI functions. So that the

AI knows how large this block is, the first word of the block always contains the number of bytes remaining in the block after the count (similar to a string definition in Pascal). For example, when drawing a line from coordinate (0, 42) to (511, 13), the parameter block for the HLINE function would be set up as in Figure 4-1.

Figure 4-1: Parameter block for the HLINE function

Parameter Block	Byte Offset	Description
8	0	Number of bytes left in block
0	2	X1
42	4	Y1
511	6	X2
13	7	Y2

Some AI functions, such as HOPEN, return information to the caller. Because the block is really still in the caller's memory, the AI needs only to modify the blocks contents in order to "return" information. Other functions, such as HCLOSE, do not require a parameter block because no information other than the call itself needs to be transferred.

Coordinates

As with many PC-based graphics boards, the drawing origin on the 8514/A is the upper left corner of the display. For example, in the 8514/A's 640 x 480 mode, this means that the upper left corner is referenced by coordinate (0, 0) and the lower right by (639, 479). The AI, through the capabilities of the 8514/A, also allows the use of coordinates outside the display area. This can be useful when you are interactively moving large blocks of data or even cursors around, because the edges of the object can go beyond the display area without requiring the application to do any clipping. Table 4-1 shows the extents of the coordinate values for the various modes.

Table 4-1: Upper- and lower-coordinate limits

Min. Coor.[1]	Max. Coor.	Min. Disp.[2]	Max. Disp.
640 x 480	–512, –256	1,536, 767	0, 0 639, 479
1,024 x 768	–512, –512	1,536, 1,536	0, 0 1,023, 767

	Min. Drawable[3]	Max. Drawable
640 x 480	0, 0	1,024, 512[4]
1,024 x 768	0, 0	1,024, 1024

1. This refers to the minimum and maximum coordinates that the AI can process.
2. These are the coordinate ranges for displayable information.
3. Off-screen memory is available in both display modes. These values reflect the extents of all drawable memory for the given mode. The off-screen memory can be used for bit-map or font storage or for any other valid drawing function.
4. In the 640 x 480 by 256-color mode, the maximum addressable Y resolution is 1,024.

Combining the AI and 8514/A Registers

One possible way to gain the benefit of both the AI and the registers is by combining their operation. The AI is handy for initializing the board and environment, whereas accessing the registers can provide better graphics performance once the initialization is complete. This is simple to accomplish, as long as you realize that the AI may have some internal variables that reflect the state of the hardware, and if you change that state, the AI may act oddly. Generally, this shouldn't be a problem. For example, you may want the AI to initialize the board and do most of the drawing, but you want to switch quickly back to VGA mode, display some things, and switch back. With 1,024 x 768 resolution, all you would need to do is write a 6 to register 04AE8H (the Advanced Function Control register) to switch to VGA mode and write a 7 to switch back. This is a lot easier than suspending the AI with the HCLOSE function and then reopening it nondestructively with HOPEN.

This combined AI-register approach does have some drawbacks, though. One is that you must have the AI loaded in memory, which may take valuable RAM from your application. The second drawback, much more major, is that there's no guarantee you're running on an 8514/A-register-compatible board. The AI is supported on several TI 34010 boards, as well as IBM's ImageAdapter/A (mentioned earlier). But if you know your application will be running only on a specific hardware configuration, the combined approach may be a good compromise.

AI Speed

Although our experiments with the IBM 8514/A have shown its AI to be half as fast as direct register access, the clone companies may have a different story to tell.

The IBM AI definitely has some inefficient portions, which many clone companies claim to have eliminated. The other thing to note is that the clone AIs take advantage of the special extensions that their particular version of the chip set incorporates. This includes such things as fast line drawing, which require substantial setup on a regular 8514/A but which may be very quick on a clone. Because the AI masks that capability of the clone hardware, your AI programs on the clone will run faster than on the IBM board and, conceivably, faster than a register-based version of the program on the IBM board as well.

This means that neglecting the AI may not be a good idea. For optimum results, try to support both the AI and the register set.

The AI Commands

What follows is a summary of each AI command, in alphabetical order, including its parameter block and any special information pertinent to the use of the function. Please note that we are using only the AI C binding function names. Appendix B lists the function numbers.

Each summary starts with the name of the function, followed by a brief overview of what the function does. Next is the description of the parameter block used by the function.

The parameter information consists of three fields: **Byte Offset**, which is the index into the parameter block; **Parameter Type & Value**, which indicates the type of parameter (BYTE = 1 byte, WORD = 2 bytes, DWORD = 4 bytes, and FARPTR = 4 bytes in segment:offset notation) and its value if constant; and **Description**, which briefly describes the parameter.

Finally, there is a detailed function description and a reference to other pertinent instructions.

Please note that all reserved parameters and bit fields should be explicitly set to 0 to avoid incompatibility with future versions of the AI.

Table 4-2 contains a list of the AI commands in alphabetical order; Table 4-3 contains the same commands grouped by function type.

If you're interested in seeing what an AI program looks like, refer to Listing 5-4 and all following ones.

Table 4-2: AI commands in alphabetical order

ABLOCKCGA	Draw block of characters (CGA-encoded)
ABLOCKMFI	Draw block of alphanumeric characters (MFI-encoded)
ACURSOR	Set character cursor position
AERASE	Erase block of characters
ASCELL	Set default character cell size
ASCROLL	Scroll block of characters
ASCUR	Set character cursor shape
ASFONT	Set alphanumeric character set
AXLATE	Assign alphanumeric color index table
HBAR	Begin area
HBBC	Bit block — copy block on-board
HBBCHN	Bit block — transfer (chain) data
HBBR	Bit block — read image data from board
HBBW	Bit block — write image data from host
HCBBW	Bit block — write at current position from host
HCCHST	Draw string at current position
HCHST	Draw string at given position
HCLINE	Draw line(s) at current position, with absolute coordinates
HCLOSE	Close adapter
HCMRK	Draw marker at current position
HCRLINE	Draw line(s) at current position, with relative coordinates
HEAR	End area
HEGS	Erase graphics screen
HESC	Escape — abort current operation
HINIT	Initialize state
HINT	Wait for event
HLDPAL	Load palette
HLINE	Draw line(s) at given position, with absolute coordinates
HMRK	Draw marker at given position
HOPEN	Open adapter

HQCOORD	Query coordinate type
HQCP	Query current position
HQDFPAL	Query default palette
HQDPS	Query drawing process state size
HQMODE	Query mode
HQMODES	Query adapter modes
HRECT	Draw filled rectangle
HRLINE	Draw line(s) at given position, with relative coordinates
HRLPC	Restore line pattern count
HRPAL	Restore palette
HSBCOL	Set background color
HSBP	Set bit-plane controls
HSCMP	Set Color Compare register
HSCOL	Set foreground color
HSCOORD	Set coordinate type
HSCP	Set current position
HSCS	Set text character set
HSGQ	Set graphics quality
HSHS	Set clipping rectangle
HSLPC	Save line pattern count
HSLT	Set line type
HSLW	Set line width
HSMARK	Set marker shape
HSMODE	Set mode
HSMX	Set mix
HSPAL	Save palette
HSPATT	Set pattern shape
HSPATTO	Set pattern reference point
HSYNC	Synchronize adapter
HXLATE	Assign multiplane text color index table

Table 4-3: AI commands grouped by function

Alphanumeric Functions

ABLOCKCGA	Draw block of characters (CGA-encoded)
ABLOCKMFI	Draw block of alphanumeric characters (MFI-encoded)
ACURSOR	Set character cursor position
AERASE	Erase block of characters
ASCELL	Set default character cell size
ASCROLL	Scroll block of characters
ASCUR	Set character cursor shape
ASFONT	Set alphanumeric character set
AXLATE	Assign alphanumeric color index table

Drawing Functions

HBAR	Begin area
HBBC	Bit block — copy block on-board
HBBCHN	Bit block — transfer (chain) data
HBBR	Bit block — read image data from board
HBBW	Bit block — write image data from host
HCBBW	Bit block — write at current position from host
HCCHST	Draw string at current position
HCHST	Draw string at given position
HCLINE	Draw line(s) at current position, with absolute coordinates
HCMRK	Draw marker at current position
HCRLINE	Draw line(s) at current position, with relative coordinates
HEAR	End area
HEGS	Erase graphics screen
HLINE	Draw line(s) at given position, with absolute coordinates
HMRK	Draw marker at given position
HRECT	Draw filled rectangle
HRLINE	Draw line(s) at given position, with relative coordinates

Control Functions

HCLOSE	Close adapter
HESC	Escape — abort current operation
HINIT	Initialize state
HINT	Wait for event
HOPEN	Open adapter
HSMODE	Set mode
HSCS	Set text character set
HSYNC	Synchronize adapter

Palette Functions

HLDPAL	Load palette
HRPAL	Restore palette
HSPAL	Save palette

Query Functions

HQCOORD	Query coordinate type
HQCP	Query current position
HQDFPAL	Query default palette
HQDPS	Query drawing process state size
HQMODE	Query mode
HQMODES	Query adapter modes

Drawing-Environment Functions

HRLPC	Restore line pattern count
HSBCOL	Set background color
HSBP	Set bit plane controls
HSCMP	Set Color Compare register
HSCOL	Set foreground color
HSCOORD	Set coordinate type
HSCP	Set current position
HSGQ	Set graphics quality
HSHS	Set clipping rectangle
HSLPC	Save line pattern count
HSLT	Set line type
HSLW	Set line width
HSMARK	Set marker shape
HSMX	Set mix
HSPATT	Set pattern shape
HSPATTO	Set pattern reference point
HXLATE	Assign multiplane text color index table

ABLOCKCGA

Write a rectangular block of characters to the display using CGA-type attributes.

Byte Offset	Parameter Type & Value	Description
0	WORD = 10	Size of remaining parameter block
2	BYTE	Initial column
3	BYTE	Initial row
4	BYTE	Width of block (in characters)
5	BYTE	Height of block (in characters)
6	FARPTR	Pointer to block in PC memory
10	BYTE	Pitch of block in PC memory (in characters)
11	BYTE	AI highlight attribute for block

This function displays a block of characters, each with its own foreground and background color, on the 8514/A display.

The initial row and column are based on the character cell height, as defined by a previous call to HSMODE or ASCELL, and refer to the position at which the block of characters is placed. The row and column number start with (0, 0) being the upper left-hand corner of the display, with the row and column increasing as you go down and right, respectively.

The width and height of the block tell the AI the dimensions of the block, both for purposes of display and for reading the character information from PC memory.

The pointer to the block is a PC address that contains the information for the first character to be displayed.

The pitch of the block is the number of characters between the first character in one row and the first character of the subsequent row. For example, the pitch of a regular 80 by 25 text display is 80. This allows for the display of a block of text that is a subset of a larger configuration.

The highlight attribute byte is used to modify the display of the characters in the block in a global fashion and is defined as follows:

Bit 7	Underline
Bit 6	Reverse video
Bit 5	Overstrike
Bit 4	1 = set background transparent 0 = set background opaque (use background color)
Bit 3	Reserved
Bit 2	Reserved
Bit 1	Font selection bit 0
Bit 0	Font selection bit 1

The character-block information is set up in a manner similar to that a normal CGA-type (color) text screen might use, which is one byte to indicate the ASCII value of the character and one byte for the attribute. The attribute byte is defined as:

Bits 7 – 4	Background color
Bits 3 – 0	Foreground color

To see how these color values relate to the normal color values, refer to the AXLATE function.

Related functions: ABLOCKMFI, ASCELL, ASFONT, AXLATE, and HSMODE.

ABLOCKMFI

Write a rectangular block of characters to the display using an AI-specific format.

Byte Offset	Parameter Type & Value	Description
0	WORD = 9	Size of remaining parameter block
2	BYTE	Initial column
3	BYTE	Initial row
4	BYTE	Width of block (in characters)
5	BYTE	Height of block (in characters)
6	FARPTR	Pointer to block in PC memory
10	BYTE	Pitch of block in PC memory (in characters)

This function displays a block of characters, each with its own foreground and background color and highlight attribute.

The initial row and column are based on the character cell height, as defined by a previous call to HSMODE or ASCELL, and refer to the position at which the block of characters is placed. The row and column number start with (0, 0) being the upper left-hand corner of the display, with the row and column increasing as you go down and right, respectively.

The width and height of the block tell the AI the dimensions of the block, both for purposes of display and for reading the character information from PC memory.

The pointer to the block refers to a PC address that contains the information for the first character to be displayed.

The pitch of the block is the number of characters between the first character in one row and the first character of the subsequent row. For example, the pitch of a regular 80 by 25 text display is 80. This allows for the display of a block of text that is a subset of a larger configuration.

The character-block information consists of four bytes of data for each character. The four bytes are defined as follows:

Byte 0	ASCII character code
Byte 1	Color attribute
	Bits 7 – 4 = background color
	Bits 3 – 0 = foreground color
Byte 2	Highlight attribute
	(see ABLOCKCGA for definition)
Byte 3	Reserved

Related functions: ABLOCKCGA, ASCELL, ASFONT, AXLATE, and HSMODE.

ACURSOR

Position the alphanumeric text cursor.

Byte Offset	Parameter Type & Value	Description
0	WORD = 2	Size of remaining parameter block
2	BYTE	Column of new cursor position
3	BYTE	Row of new cursor position

This function positions a cursor on the display. The cursor positioning is based on the character cell size defined by HSMODE or ASCELL. The cursor is drawn in the XOR mode so that it will always be visible.

Related functions: ASCELL, ASCUR, and HSMODE.

AERASE

Erase a block of characters.

Byte Offset	Parameter Type & Value	Description
0	WORD = 5	Size of remaining parameter block
2	BYTE	Initial column
3	BYTE	Initial row
4	BYTE	Width of block (in characters)
5	BYTE	Height of block (in characters)
6	BYTE	Color

This function clears a rectangular region of the display to a specific color. It is meant to be used in conjunction with the alphanumeric text functions.

The initial row and column are based on the character cell height, as defined by a previous call to HSMODE, and refer to the position at which the block of characters is placed. The row and column number start with (0, 0) being the upper left-hand corner of the display, with the row and column increasing as you go down and right, respectively.

The width and height of the block tell the AI the dimensions of the block so that it knows how much to clear.

The color byte is defined as follows:

Bits 7 – 4	Background color
Bits 3 – 0	Reserved

Related functions: ASCELL, AXLATE, HRECT, and HSMODE.

ASCELL

Set the cell size for alphanumeric text.

Byte Offset	Parameter Type & Value	Description
0	WORD = 4	Size of remaining parameter block
2	BYTE	New cell width
3	BYTE	New cell height

This function changes the alphanumeric cell size defined by an earlier call to HSMODE or ASCELL. The horizontal and vertical character dimensions, in characters, of the display can be calculated by dividing the screen resolution by the appropriate font cell dimension.

Specifying a value of 0 for the height or width will result in the current value being used for that dimension instead.

If you do change the default cell size for the current mode, you must also load a new font that matches the newly defined cell size prior to doing any other alphanumeric calls or you may get strange results.

Related functions: ASFONT and HSMODE.

ASCROLL

Copy a rectangular block of characters on the display to another position on the display.

Byte Offset	Parameter Type & Value	Description
0	WORD = 6	Size of remaining parameter block
2	BYTE	Initial column
3	BYTE	Initial row
	BYTE	Width of block (in characters)
5	BYTE	Height of block (in characters)
6	BYTE	Destination column
7	BYTE	Destination row

This function copies a block of characters starting at the initial column and row to the destination column and row. The block dimensions are defined by the width and height fields.

The row and column positions are based on the character cell height as defined by a previous call to HSMODE. The row and column numbers start with (0, 0) being the upper left-hand corner of the display, with the row and column increasing as you go down and right, respectively.

The coordinates and dimension must be specified so as to ensure that all copying occurs entirely on the screen — that is, both the original and destination blocks must be visible. The blocks can overlap.

Related functions: ASCELL and HSMODE.

ASCUR

Define the alphanumeric cursor's shape and attributes.

Byte Offset	Parameter Type & Value	Description
0	WORD = 3	Size of remaining parameter block
2	BYTE	Cursor start line
3	BYTE	Cursor stop line
4	BYTE	Cursor attribute

This function sets the cursor position and size within the cursor cell, as well as its attribute. The cursor cell size is the height of the current font as selected by HSMODE.

The cursor start line defines the line within the cursor cell at which the block cursor should start. The top of the cursor cell is 0. The cursor stop line specifies the last line of the block cursor relative to the cursor cell. Note: If the cursor stop line is less than the cursor start line, no cursor is displayed unless the start line is defined to be 255 (0xFF), in which case the previous cursor start- and end-line information is retained.

The cursor attribute specifies the type of cursor or the cursor state:

Attribute	Description
0	Normal block cursor
1	Hidden — this supersedes the start/stop values
2	Left arrow
3	Right arrow

Related functions: ACURSOR, ASCELL, and HSMODE.

ASFONT

Define a character set for use with the alphanumeric functions.

Byte Offset	Parameter Type & Value	Description
0	WORD = 6	Size of remaining parameter block
2	BYTE	Font number to use
3	BYTE	Reserved
4	FARPTR	Address of the font definition

This function defines font numbers for use with the alphanumeric text functions ABLOCKCGA and ABLOCKMFI.

The font number can range in value from 0 to 3 and is used primarily as the font handle for text operations.

The font definition, whose address is required by this function, can be one of two different font types: monochrome bit-mapped and short-stroke vector. For an example of how to use the latter with the regular text functions, refer to Chapter 5. For more detailed information about the font structure and definitions, refer to Appendix C.

Related functions: ABLOCKCGA, ASCELL, ASFONT, HSCS, and HSMODE.

AXLATE

Define alphanumeric color translation tables.

Byte Offset	Parameter Type & Value	Description
0	WORD = 128	Size of remaining parameter block
2	DWORD	16 foreground colors
66	DWORD	16 background colors

This function sets up a translation table to translate the foreground and background colors used by the ABLOCKCGA, ABLOCKMFI, and AERASE functions to AI-specific colors. Note that each entry is defined as a 32-bit entry. This is to maintain compatibility with future versions of the AI that might be able to support 32-bit color.

The translation mechanism uses the 4-bit foreground or background color as an index into the foreground or background color table defined by this function, and the color from the table is the one actually used to display the text.

Related functions: ABLOCKCGA, ABLOCKMFI, AERASE, and HLDPAL.

HBAR

Begin area definition.

Byte Offset	Parameter Type & Value	Description
0	WORD = 0	Size of remaining parameter block

No parameters are required.

This function tells the AI to start using subsequent drawing commands to define an area on the display that can be used as a boundary for an area fill. This is the way you draw filled polygons with the AI if you don't want to implement your own polygon-rendering algorithm. The definition process is completed or suspended by using the HEAR function.

You must observe several things when you use this function. First, the boundary definition can consist of one or more closed graphical objects.

A closed object consists of a contiguous series of lines. The definition of the first closed object starts with the first drawing command subsequent to the HBAR command. The definition of the first closed object is finished when a HEAR command or move command (that is, nondrawing command that moves the current drawing position) is received. The AI ensures that the object is closed by drawing a line from the last point defined to the first one. This process can go on ad infinitum with the same HBAR. Area definitions cannot be nested.

Once the HEAR command to fill the defined area(s) has been received, the AI proceeds to do the fill using the classical odd/even-fill technique. One way to look at this is by envisioning the whole area definition to be bounded by a rectangle. When scanning each line of the rectangle at a time, the filling is turned off, but as the scan

moves right, every time a line is encountered, the fill state is changed. Therefore, after each odd line found, filling is enabled, and after each even line, it is disabled. Based on this, if you picture a definition consisting of a rectangle within a rectangle, only the space between the inner and the outer rectangle would be filled.

The following functions must not be used during an area definition because they will cause unpredictable results: HCCHST, HCHST, HCLOSE, HOPEN, HSBP, HSCMP, HSHS, and HSMODE.

Other primitives that interfere with the current position might also cause problems if the current position is not properly restored prior to continuing with other drawing commands.

Also important is that the area definition will potentially use all the off-screen work space — that is, that area of the 8514/A memory that resides beyond the display portion. This means that you cannot store image data off-screen and use this command at the same time.

Related function: HEAR.

HBBC

BitBLT a rectangular region from one location to another on the board.

Byte Offset	Parameter Type & Value	Description
0	WORD = 16	Size of remaining parameter block
2	WORD	Type of BitBLT
4	WORD	Width of the block (in pixels)
6	WORD	Height of the block (in lines)
8	BYTE	Source plane for bit expansion
9	BYTE	Reserved
10	WORD	Source X
12	WORD	Source Y
14	WORD	Destination X
16	WORD	Destination Y

This function copies a block of image data on the 8514/A from one place to another using the current foreground mix (see HSMX).

The type of BitBLT indicates whether this is a straight copy of all planes (that is, all 4 bits per pixel or 8 bits per pixel, depending on your board configuration) or a monochrome-to-color expansion (or single-plane source) BitBLT. The expansion BitBLT is useful for quickly displaying bit-mapped fonts because most of these font definitions have bits turned on or off. When this type of operation has been selected, the foreground color and mix are used for all bits in the source-plane rectangle that are set to 1, and for all bits set to 0, the background color and mix are used. The type of operation is specified by the following values:

Value	Operation
0	Expand a single plane
8	Copy all planes

The source plane for the bit expansion specifies which plane to use in the case of an expansion BitBLT. This parameter has no meaning in the all-plane BitBLT.

The width and height of the block tell the AI the dimensions of the block to be copied.

The source and destination coordinates refer to the upper left corner pixel of the blocks to be manipulated. Overlapping blocks are handled correctly — the AI always puts the source information in the specified destination block area.

Related functions: HSCOL, HSBCOL, HSHS, and HSMX.

HBBCHN

Chain BitBLT data for board-PC transfer.

Byte Offset	Parameter Type & Value	Description
0	WORD = 6	Size of remaining parameter block
2	FARPTR	Pointer to data buffer for transfer
6	WORD	Size of buffer

This function specifies a data buffer to be used for transferring data to or from the 8514/A, as specified by a prior HBBR, HBBW, or HCBBW command.

Multiple HBBCHN commands are generally used to effect the transfer of a whole image. If used in conjunction with rectangular PC data instead of linear data (defined by the three functions mentioned earlier), then the data pointed at by this function must be in rectangular format, with an appropriate size.

The current position is not affected by this function.

Related functions: HBBR, HBBW, and HCBBW.

HBBR

Read image data from 8514/A (BitBLT from board).

Byte Offset	Parameter Type & Value	Description
0	WORD=12 or 20	Size of remaining parameter block
2	WORD	Type of BitBLT
4	WORD	Width of the block (in pixels)
6	WORD	Height of the block (in lines)
8	BYTE	Source plane for read
9	BYTE	Reserved
10	WORD	Source X
12	WORD	Source Y
14	WORD	X offset
16	WORD	Y offset
18	WORD	Destination pitch
20	WORD	Destination height

This function initiates a BitBLT on the 8514/A that results in image data being transferred to the PC via the HBBCHN function.

The type of BitBLT indicates whether this is a straight read of all planes (that is, all 4 bits per pixel or 8 bits per pixel, depending on the board configuration) or a single-plane read. The type of operation is specified by the following values:

Value	Operation
0	Read a single plane
8	Read all planes

The source plane for read specifies which plane to use in the case of a single-plane read. This parameter has no meaning in the all-plane read.

The width and height of the block tell the AI the dimensions of the block to be read.

The source X and Y specify the upper left corner of the block to be read on the 8514/A.

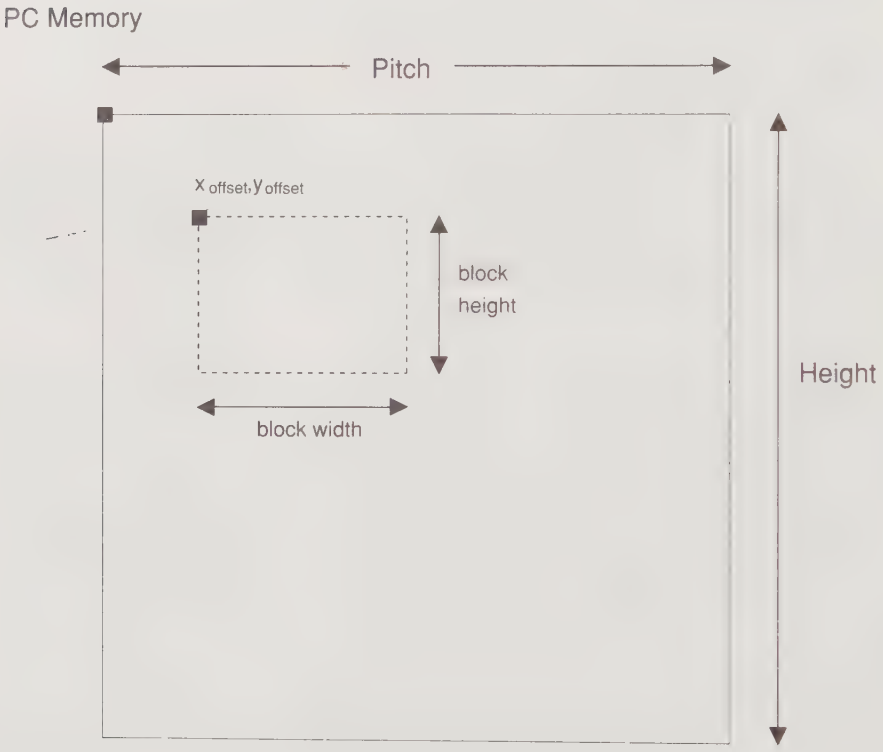
In the all-plane read, each pixel gets a byte of its own, even if you're dealing only with a 4-bit-per-pixel 8514/A. In the single-plane read, the bits are packed 8 to a byte, and the first bit of each line read starts on a byte boundary. The end of the line is padded with 0 bits if the line width is not a multiple of 8.

If the parameter-size field is 20, the AI will copy the image data into a rectangular region, instead of just a linear region, in PC memory. The X offset specifies, in pixels, the X offset into a rectangular region in memory, and the Y offset does the same for the Y, but in lines. The destination pitch specifies the width of the rectangle into which the data is being imported. The destination height specifies the height of the block in memory. See Figure 4-2 for a graphical explanation.

The current position is set to the source X and Y positions.

Related function: HBBCHN.

Figure 4-2: Addressing PC memory using a rectangular coordinate system



HBBW

Write image data from PC to 8514/A (BitBLT to board).

Byte Offset	Parameter Type & Value	Description
0	WORD=10 or 18	Size of remaining parameter block
2	WORD	Type of BitBLT
4	WORD	Width of the block (in pixels)
6	WORD	Height of the block (in lines)
8	WORD	Destination X
10	WORD	Destination Y
12	WORD	X offset
14	WORD	Y offset
16	WORD	Source pitch
18	WORD	Source height

This function initiates a BitBLT on the 8514/A that results in image data being transferred from the PC via the HBBCHN function.

The type of BitBLT indicates whether this is a straight write of all planes (that is, all 4 bits per pixel or 8 bits per pixel, depending on the board configuration) or a single-plane write. The type of operation is specified by the following values:

Value	Operation
0	Monochrome-to-color expansion write
8	Write all planes

The width and height of the block tell the AI the dimensions of the block to be written.

The destination X and Y specify the upper left corner of the block to be written on the 8514/A.

In the all-plane write, each source pixel must be in a byte of its own. The foreground mix is used for this operation.

In the expansion write, the source bits are packed 8 to a byte and the first bit of each line read starts on a byte boundary. Each source bit that is set to 1 is expanded into the foreground color using the foreground mix, and each 0 bit with the background color and background mix.

If the parameter size field is 18, the AI will copy the image data from a rectangular region, instead of just a linear region, in PC memory. The X offset specifies, in pixels, the X offset into a rectangular region in memory, and the Y offset does the same for the Y, but in lines. The destination pitch specifies the width of the rectangle into which the data is being imported. The destination height specifies the height of the block in memory. See Figure 4-2 for a graphical explanation.

The current position is set to destination X and Y.

Related functions: HBBCHN, HSCOL, HSBCOL, HSHS, and HSMX.

HCBBW

Write image data from PC to 8514/A (BitBLT to board) at current position.

Byte Offset	Parameter Type & Value	Description
0	WORD = 6 or 14	Size of remaining parameter block
2	WORD	Type of BitBLT
4	WORD	Width of the block (in pixels)
6	WORD	Height of the block (in lines)
8	WORD	X offset
10	WORD	Y offset
12	WORD	Source pitch
14	WORD	Source height

This function initiates a BitBLT on the 8514/A that results in image data being transferred from the PC via the HBBCHN function. The only difference between this function and HBBW is that the current position is used as the destination of the block write.

The type of BitBLT indicates whether this is a straight write of all planes (that is, all 4 bits per pixel or 8 bits per pixel, depending on the board configuration) or a single-plane write. The type of operation is specified by the following values:

Value	Operation
0	Write a single plane
8	Write all planes

The width and height of the block tell the AI the dimensions of the block to be written.

The destination X and Y specify the upper left corner of the block to be written on the 8514/A.

In the all-plane write, each source pixel must be in a byte of its own. The foreground mix is used for this operation.

In the expansion write, the source bits are packed 8 to a byte and the first bit of each line read starts on a byte boundary. Each source bit that is set to 1 is expanded into the foreground color using the foreground mix, and each 0 bit with the background color and background mix.

If the parameter size field is 14, the AI will copy the image data from a rectangular region, instead of just a linear region, in PC memory. The X offset specifies, in pixels, the X offset into a rectangular region in memory, and the Y offset does the same for the Y, but in lines. The destination pitch specifies the width of the rectangle into which the data is being imported. The destination height specifies the height of the block in memory.

Related functions: HBBCHN, HSCOL, HSBCOL, HSHS, and HSMX.

HCCHST

Write a text string at the current position.

Byte Offset	Parameter Type & Value	Description
0	WORD >= 0	Size of remaining parameter block
2	BYTE	Character string

This function displays a text string in the current font (selected by HSCS) at the current drawing position. The length of the text string is determined by the parameter-block size. The current position references the lower left corner of the first character in the string.

The current position is not altered by this function.

Related functions: HCHST, HSBCOL, HSCS, HSCOL, HSHS, and HSMX.

HCHST

Write a text string at the given position.

Byte Offset	Parameter Type & Value	Description
0	WORD >= 4	Size of remaining parameter block
2	WORD	X position
4	WORD	Y position
6	BYTE	Character string

This function displays a text string in the current font (selected by HSCS) at the given drawing position. The length of the text string is determined by the parameter-block size minus 4 (the size of the position parameters). The given position references the lower left corner of the first character in the string.

The current position is set to the given position.

Related functions: HCCHST, HSBCOL, HSCS, HSCOL, HSHS, and HSMX.

HCLINE

Draw a polyline starting at current position.

Byte Offset	Parameter Type & Value	Description
0	WORD >= 0	Size of remaining parameter block
2	WORD	X0
4	WORD	Y0
6	WORD	X1
8	WORD	Y1
.		
.		
.		
n*4 + 2	WORD	Xn
n*4 + 4	WORD	Yn

This function draws a polyline, starting at the current drawing position, to (X0, Y0), from there to (X1, Y1), and so on. The number of points is determined by the size of the parameter block.

The current position is set to the last point in the polyline. This function does not reset the line pattern.

Related functions: HCRLINE, HLINE, HRLINE, HSBCOL, HSCOL, HSHS, HSLT, HSLW, and HSMX.

HCLOSE

Close down the 8514/A.

Byte Offset	Parameter Type & Value	Description
0	WORD = 0	Size of remaining parameter block

No parameters are required.

This function disables the display of the 8514/A on the monitor attached to it and reenables the VGA loop-through so that the monitor displays only VGA information.

Related function: HOPEN.

HCMRK

Draw a marker at current position and all others specified.

Byte Offset	Parameter Type & Value	Description
0	WORD >= 0	Size of remaining parameter block
2	WORD	X0
4	WORD	Y0
6	WORD	X1
8	WORD	Y1
.		
.		
.		
n*4 + 2	WORD	Xn
n*4 + 4	WORD	Yn

This function draws a marker (as defined by the HSMARK command) at the current position. If additional points are present (determined by the size of the parameter block), then markers are drawn at the additional points as well.

The marker is drawn such that it is centered about the point specified. Note that there may be a small (one pixel) discrepancy in positioning because of the integer math being used. The formula for positioning the upper left corner of the marker is:

ULX = X – (markWidth/2)
ULY = Y – (markHeight/2)

The current position is set to the last point in the parameter block.

Related functions: HMRK, HSCOL, HSHS, HSLT, HSLW, and HSMX.

HCRLINE

Draw a polyline starting at the current position with relative coordinates.

Byte Offset	Parameter Type & Value	Description
0	WORD >= 0	Size of remaining parameter block
2	BYTE	dX0
3	BYTE	dY0
4	BYTE	dX1
5	BYTE	dY1
.		
.		
.		
n*2 + 2	BYTE	dXn
n*2 + 3	BYTE	dYn

This function draws a polyline, starting at the current drawing position (CPX, CPY), to (CPX + dX0, CPY + dY0), from there to (CPX + dX0 + dX1, CPY + dY0 + dY1), and so on. The number of points is determined by the size of the parameter block. Note that each delta value is only a byte, which limits the range for each new point to –128 to +127.

The current position is set to the last point calculated in the polyline. This function resets the line pattern prior to executing.

Related functions: HCLINE, HLINE, HRLINE, HSBCOL, HSCOL, HSHS, HSLT, HSLW, and HSMX.

HEAR

Begin area definition.

Byte Offset	Parameter Type & Value	Description
0	WORD = 1	Size of remaining parameter block
2	BYTE	End-area flags

This function ends an area definition initiated by HBAR. The flags are defined as follows:

Flag Byte	Description
0	Fill the defined area. This performs an area fill, as described in the HBAR function.
40H	Suspend the definition. Selecting this option suspends the area definition until the next HBAR command, leaving the off-screen work area untouched. The definition should continue from where it left off after the HBAR command.
80H	Abort the definition. This aborts the definition and clears the off-screen buffer used to store the definition information.

If an area fill is selected, the current color, pattern, and mix are used.

Related functions: HBAR, HSCOL, HSHS, HSPATT, HSPATTO, and HSMX.

HEGS

Clear the graphics display.

Byte Offset	Parameter Type & Value	Description
0	WORD = 0	Size of remaining parameter block

No parameters are required.

This function erases the display by setting all pixels to 0. The erase is affected by the setting of the clipping rectangle as well as the plane enable.

Related functions: HSBP and HSHS.

HESC

Escape/abort current operation.

Byte Offset	Parameter Type & Value	Description
0	WORD = 0	Size of remaining parameter block

No parameters are required.

This function does not apply to the 8514/A. It has apparently been defined for future use on other AI-compatible adapters. If called, the AI just returns.

HINIT

Initialize task data area.

Byte Offset	Parameter Type & Value	Description
0	WORD = 2	Size of remaining parameter block
2	WORD	Segment of the task buffer

This function initializes the task data structure located at offset 0000 in the segment passed to the function.

The purpose of the task buffer is to ease multitasking by allowing for swapping of these task buffers. In the DOS environment, there really isn't a whole lot of use for this, except in custom applications. But this function is a necessary step for using the AI. There are actually two types of task information: task dependent and task independent. Task-independent data consists of things such as color-palette contents and adapter mode; task-dependent data refers to more application-specific attributes such as current position, color, and so on.

When HINIT is called, the task-dependent data, and therefore the current task's environment, is initialized with the following information:

Current position	(0, 0)
Foreground color	All bits set
Background color	All bits cleared
Foreground mix	Dst = Src (Replace mode)
Background mix	Not initialized
Comparison color	Not initialized
Compare function	FALSE (everything gets drawn)
Line type	Solid
Line width	1 pixel
Line-pattern info.	Not initialized
Area pattern	Solid
Area-pattern origin:	(0, 0)
Text and marker info.	Not initialized
Clipping rectangle	Full screen
Graphics quality	High precision, last pixel null
Plane mask	All planes enabled
Color index translate table	The 8 entries are set linearly, 0 – 7
Alphanumeric cursor	At (0, 0) and hidden
Alphanumeric color translate	Two sets of 16 linear values, 0 – 15 each for foreground and background
Default font	Not initialized

Related functions: HSYNC and HQDPS.

HINT

Wait for event or interrupt.

Byte Offset	Parameter Type & Value	Description
0	WORD = 4	Size of remaining parameter block
2	DWORD	Event identifier

This function waits until the requested event occurs. The event is specified by setting the appropriate bit in the event-identifier parameter. All other nonpertinent bits should be cleared, as follows:

Bit	Event
31	Wait for vertical blank. This is useful for applications that need to time certain graphics operations — for example, to reduce flicker during animations.
30 – 0	Reserved

Upon return to the calling program, the event-identifier field will contain only the single bit set that caused the function to terminate the wait.

HLDPAL

Load a palette into the color lookup table.

Byte Offset	Parameter Type & Value	Description
0	WORD = 1 or 10	Size of remaining parameter block
2	BYTE	Palette type
3	BYTE	Reserved
4	WORD	Index of first entry
6	WORD	Number of entries
8	FARPTR	Address of data

This function loads either the default palette or an application-specific palette into the color lookup table located on the 8514/A (see Figure 5-1).

The palette-type parameter indicates whether the default or application palette is to be used, as follows:

Value	Type
0	Application specific. Use rest of parameters.
1	Default. Remaining parameters are ignored and entire default palette is used. The default AI palette is the same as that used by a CGA (see HQDFPAL), repeated through all 256 entries (16 times).

The index of the first entry specifies what part of the palette to start loading. This means that you have the ability to set only a small part of the palette if you choose to.

The number of entries allows you to specify the number of entries you would like to modify in the color lookup table .

The lookup-table data should contain four bytes for every entry you want to update. The four bytes are defined as follows:

Byte	Description
0	Red component
1	Green component
2	Blue component
3	Reserved

The values for the red, green, and blue bytes range from 0 to 255, with 255 being the highest intensity. On the IBM 8514/A, the two low-order bits are ignored because the IBM 8514/A supports only 6 bits of intensity per gun.

Note: Mode changes might corrupt the palette. You may want to use HRPAL and HSPAL to save and restore the palette while attempting a mode change.

Related functions: HRPAL and HSPAL.

HLINE

Draw a polyline starting at the specified position.

Byte Offset	Parameter Type & Value	Description
0	WORD >= 0	Size of remaining parameter block
2	WORD	X0
4	WORD	Y0
6	WORD	X1
8	WORD	Y1
.		
.		
.		
n*4 + 2	WORD	Xn
n*4 + 4	WORD	Yn

This function draws a polyline, starting at (X0, Y0), to (X1, Y1), from there to (X2, Y2), and so on. The number of points is determined by the size of the parameter block.

The current position is set to the last point in the polyline. This function does not reset the line pattern.

Related functions: HCLINE, HCRLINE, HRLINE, HSBCOL, HSCOL, HSHS, HSLT, HSLW, and HSMX.

HMRK

Draw a marker at the specified positions.

Byte Offset	Parameter Type & Value	Description
0	WORD >= 0	Size of remaining parameter block
2	WORD	X0
4	WORD	Y0
6	WORD	X1
8	WORD	Y1
.		
.		
.		
n*4 + 2	WORD	Xn
n*4 + 4	WORD	Yn

This function draws a marker (as defined by the HSMARK command) at the specified points. The number of points is determined by the size of the parameter block.

The marker is drawn such that it is centered about the point specified. Note that there may be a small (one pixel) discrepancy in positioning because of the integer math being used. The formula for positioning the upper left corner of the marker is:

ULX = X – (markWidth/2)
ULY = Y – (markHeight/2)

The current position is set to the last point in the parameter block.

Related functions: HSCOL, HSHS, HSLT, HSLW, HCMRK, and HSMX.

HOPEN

Enable the adapter for use.

Byte Offset	Parameter Type & Value	Description
0	WORD = 3	Size of remaining parameter block
2	BYTE	Initialization flags
3	BYTE	Mode
4	BYTE	Return status

This function initializes the task-independent state (see HINIT).

The initialization flags tell the AI what should be initialized and what shouldn't, as follows:

Bits	Meaning
7	Don't clear board memory if bit is set
6	Don't load default palette if bit is set
5 – 0	Reserved; set to 0

The mode specifies what mode to initialize the adapter into. The existing mode numbers are:

Mode Number	Screen Size	Alpha Text Size	Text Resolution
0	1,024 x 768	12 by 20	85 by 38
1	640 x 480	8 by 14	80 by 34
2	1,024 x 768	8 by 14	128 by 54
3	1,024 x 768	7 by 15	146 by 52

All other modes are reserved for future use.

The return status is used to indicate whether or not the initialization was successful. If a nonzero status is returned, then the initialization failed.

Related functions: HCLOSE, HINIT, and HSMODE.

HQCOORD

Query coordinate types.

Byte Offset	Parameter Type & Value	Description
0	WORD = 4	Size of remaining parameter block
2	BYTE	Absolute-coordinate format
3	BYTE	Relative-coordinate format
4	BYTE	Number of dimensions
5	BYTE	Result

This function determines the validity of an HSCOORD call prior to making it. Although it is anticipated that this call will serve some real function in future implementations of the AI, the IBM 8514/A version of the AI has only one viable setting for each parameter, as noted below.

The absolute- and relative-coordinate formats specify the amount of space consumed by absolute and relative coordinates, respectively:

Absolute

Bits	Definition
7 – 4	Number of bytes in each coordinate; default = 2
3 – 0	Number of fractional bytes per coordinate; default = 0

Relative

Bits	Definition
7 – 4	Number of bytes in each coordinate; default = 1
3 – 0	Number of fractional bytes per coordinate; default = 0

The number of dimensions can be two, three, or four. The 8514/A AI currently supports only two dimensions.

The result field is set based on the interpretation of the other parameters by the AI, as follows:

Bit	Definition
7	If set, the absolute-coordinate format requested is not supported
6	If set, the relative-coordinate format requested is not supported
5	If set, the number of dimensions requested is not supported
4 – 0	Reserved

Related function: HSCOORD.

HQCP

Query current position.

Byte Offset	Parameter Type & Value	Description
0	WORD = 4	Size of remaining parameter block
2	WORD	Current X
4	WORD	Current Y

After this function is called, the current position is returned in the appropriate field in the parameter block.

Many of the drawing functions alter the current position and are documented as such.

Related function: HSCP.

HQDFPAL

Query default palette.

Byte Offset	Parameter Type & Value	Description
0	WORD = 64	Size of remaining parameter block
2	DWORD	Color index values (16)

This function returns the 16 color index values that correspond to the following list of colors when the default palette is used:

Index	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	White
8	Grey
9	Light blue
10	Light green
11	Light cyan
12	Light red
13	Light magenta
14	Yellow
15	Bright white

Related functions: HLDPAL, HRPAL, and HSPAL.

HQDPS

Query task state and driver stack usage sizes.

Byte Offset	Parameter Type & Value	Description
0	WORD = 6	Size of remaining parameter block
2	WORD	Task buffer size
4	WORD	Stack usage
6	WORD	Palette save area size

(See Figure 5-1 for a complete default palette listing.)

The AI returns the size, in bytes, of several AI-related data areas. This function is called prior to HINIT so that the application can properly allocate memory for the task-dependent information, stack, and palette.

The task buffer size reflects how much memory the task-dependent information will occupy.

The stack usage value indicates the maximum amount of stack used by the AI driver.

The palette save area size tells the application how much memory it should allocate in order to use the HSPAL and HRPAL functions.

Related functions: HINIT, HRPAL, and HSPAL.

HQMODE

Query current mode and configuration.

Byte Offset	Parameter Type & Value	Description
0	WORD = 18	Size of remaining parameter block
2	BYTE	Mode number
3	WORD	Driver code value
5	BYTE	Adapter type
6	BYTE	Display type (reserved)
7	BYTE	Alphanumeric cell width (pixels)
8	BYTE	Alphanumeric cell height (pixels)
9	BYTE	Number of planes
10	WORD	Screen width
12	WORD	Screen height
14	WORD	Pixels/inch — horizontal
16	WORD	Pixels/inch — vertical
18	BYTE	Monochrome or color
19	BYTE	Intensity levels

This function returns information about the current mode and configuration of the 8514/A.

The mode number is one of the following (see HOPEN for more information):

Mode Number	Screen Size
0	1,024 x 768
1	640 x 480
2	1,024 x 768
3	1,024 x 768

The driver code value reflects the AI release number, planar configuration of the board, and whether or not the driver is running in protected mode or not, as follows:

Bits	Definition
15 – 7	Reserved
6	0 = real mode, 1 = protected mode
5	0 = 4-plane board, 1 = 8-plane board
4 – 0	Release number (starts at 0)

The adapter type for an 8514/A is always 3.

The display type is reserved and set to 0.

The alphanumeric cell width and height are dependent on the last HSMODE or ASCELL command.

The number of bit planes depends on the board's memory configuration; valid values are 4 or 8.

The screen width and height are specified in pixels and depend on the latest HOPEN or HSMODE call.

The number of pixels per inch depends on the monitor attached to the 8514/A and the resolution currently being displayed (the pixels per inch in all three cases apply to both horizontal and vertical dimensions), as follows:

Pixels/Inch	Monitor	Resolution
59	IBM 8514	640 x 4 80
79	IBM 8512 or 8513	640 x 480
92	IBM 8514	1,024 x 768

The monochrome or color field is set to 0 for a monochrome display or 255 (0xFF) for a color display.

The intensity-levels field reflects the number of intensity bits per gun in the palette. A value of 0 indicates nonloadable palettes. The normal value for an IBM 8514/A is 6.

Related functions: ASCELL, HOPEN, and HSMODE.

HQMODES

Query available modes.

Byte Offset	Parameter Type & Value	Description
0	WORD = 33	Size of remaining parameter block
2	BYTE	Adapter type
3	BYTE	First mode number
4	BYTE	Second mode number
.		
.		
.		
34	BYTE	32nd mode number

This function returns information about the available modes for the current adapter.

The adapter type returned is 3 for the 8514/A.

The mode numbers are returned in a list containing up to 32 mode numbers. When there are less than 32 valid modes for the current adapter/monitor configuration, the list is terminated by a mode number with a value of 255 (0xFF). For example, with an IBM 8514 monitor attached, all four mode numbers that the board supports are listed: 0, 1, 2, 3, 255. For the IBM 8512 and 8513, only one mode is listed: 1, 255. The resolutions of the modes are listed under HSMODE or HOPEN.

Related functions: HOPEN and HSMODE.

HRECT

Draw a filled rectangle.

Byte Offset	Parameter Type & Value	Description
0	WORD = 8	Size of remaining parameter block
2	WORD	X
4	WORD	Y
6	WORD	Width of the rectangle (in pixels)
8	WORD	Height of the rectangle (in lines)

This function draws a rectangle filled with the current area pattern. Only monochrome patterns are supported.

The X and Y coordinates specify the upper left corner of the rectangle, and the width and height specify the dimensions.

The current position is set to (X, Y.)

Related functions: HSCOL, HSHS, HSMX, HSPATT, and HSPATTO.

HRLINE

Draw a polyline starting at the specified position with relative coordinates.

Byte Offset	Parameter Type & Value	Description
0	WORD >= 0	Size of remaining parameter block
2	WORD	X
4	WORD	Y
6	BYTE	dX0
7	BYTE	dY0
8	BYTE	dX1
9	BYTE	dY1
.		
.		
.		
n*2 + 6	BYTE	dXn
n*2 + 7	BYTE	dYn

This function draws a polyline, starting at (X, Y), to (X + dX0, Y + dY0), from there to (X + dX0 + dX1, Y + dY0 + dY1), and so on. The number of points is determined by the size of the parameter block. Note that each delta value is only a byte, which limits the range for each new point to –128 to +127.

The current position is set to the last point calculated in the polyline. This function resets the line pattern prior to executing.

Related functions: HCLINE, HLINE, HRLINE, HSBCOL, HSCOL, HSHS, HSLT, HSLW, and HSMX.

HRLPC

Restore a previously saved line-pattern count.

Byte Offset	Parameter Type & Value	Description
0	WORD = 0	Size of remaining parameter block

No parameters are required.

This function restores the line-pattern count that was previously saved by the HSLPC command.

The line-pattern count is a pointer into the line bit pattern. As pixels in a line are drawn, the index is incremented until the end of the pattern is reached, in which case it is reset to the beginning of the pattern.

If this function is called without a prior HSLPC, the default value is used.

Related function: HSLPC.

HRPAL

Restore a previously saved palette.

Byte Offset	Parameter Type & Value	Description
0	WORD = Special	Size of remaining parameter block
2	BYTE	Palette data

This functions restores the palette located in the parameter block to the board. The size of the palette save area is determined by a call to HQDPS. The palette is saved in the save area by the HSPAL call.

Related function: HSPAL.

HSBCOL

Set the background color.

Byte Offset	Parameter Type & Value	Description
0	WORD = 4	Size of remaining parameter block
2	DWORD	Color

This function sets the background color on the 8514/A. Note that even though the 8514/A supports only 4 or 8 bits per pixel, the color value can be up to 32 bits long.

Related function: HSCOL.

HSBP

Enable/disable bit planes for updating and displaying.

Byte Offset	Parameter Type & Value	Description
0	WORD = 12	Size of remaining parameter block
2	DWORD	Plane enable for graphics
6	DWORD	Plane enable for alphanumerics
10	DWORD	Plane enable for display

This function disables planes from being written or displayed. Each bit in the plane determines the access or display of a plane. In a 4-bit-per-pixel configuration, only the low-order 4 bits are valid, and in an 8-bit-per-pixel system, the low-order 8 bits are valid. If the bit corresponding to a particular plane is set, access to that plane is enabled. This function overrides all other graphical modifiers, including mixes and color compare.

The plane enable for graphics controls all writes to the graphics board's memory for drawing and regular text operations.

The plane enable for alphanumerics controls all writes to display memory for all alphanumeric operations.

The plane enable for display controls which planes are displayable on the attached monitor.

Note that these plane enables in no way prevent you from reading the contents of display memory and they all default to having all valid planes enabled.

HSCMP

Set color-compare information.

Byte Offset	Parameter Type & Value	Description
0	WORD = 56	Size of remaining parameter block
2	DWORD	Color-compare color
6	BYTE	Compare logic

This functions sets the color-compare color and compare logic. The purpose of the color compare to is to provide a diverse mechanism that allows certain color values in display memory to be excluded from update, based on a color value and a logical comparison with that color. If the result of the operation is TRUE, then the pixel in memory is left unmodified.

The compare-logic field contains information about what type of comparison should be performed with the color-compare color (CC) and the destination pixel's color. The operations are defined as follows:

Value	Operation
0	TRUE
1	Destination > CC
2	Destination == CC
3	Destination < CC
4	FALSE
5	Destination >= CC
6	Destination != CC
7	Destination <= CC

Color compares do not work with alphanumeric commands.

HSCOL

Set the foreground color.

Byte Offset	Parameter Type & Value	Description
0	WORD = 4	Size of remaining parameter block
2	DWORD	Color

This function sets the foreground color on the 8514/A. Note that even though the 8514/A supports only 4 or 8 bits per pixel, the color value can be up to 32 bits long.

Related function: HSBCOL.

HSCOORD

Set coordinate types.

Byte Offset	Parameter Type & Value	Description
0	WORD = 3	Size of remaining parameter block
2	BYTE	Absolute-coordinate format
3	BYTE	Relative-coordinate format
4	BYTE	Number of dimensions

This function sets the coordinate type for the adapter to use. The validity of the request should be determined by first calling HQCOORD with the data in the parameter block.

The absolute- and relative-coordinate formats specify the amount of space consumed by absolute and relative coordinates, respectively:

Absolute

Bits	Definition
7 – 4	Number of bytes in each coordinate; default = 2
3 – 0	Number of fractional bytes per coordinate; default = 0

Relative

Bits	Definition
7 – 4	Number of bytes in each coordinate; default = 1
3 – 0	Number of fractional bytes per coordinate; default = 0

The number of dimensions can be two, three, or four. The 8514/A AI currently supports only two dimensions.

The result field is set based on the interpretation of the other parameters by the AI, as follows:

Bit	Definition
7	If set, the absolute-coordinate format requested is not supported
6	If set, the relative-coordinate format requested is not supported
5	If set, the number of dimensions requested is not supported
4 – 0	Reserved

In the current 8514/A AI, this call is ignored.

Related function: HQCOORD.

HSCP

Set current position.

Byte Offset	Parameter Type & Value	Description
0	WORD = 4	Size of remaining parameter block
2	WORD	New X position
4	WORD	New Y position

This function sets the current position for drawing functions to (X, Y).

Related function: HQCP.

HSCS

Define character set.

Byte Offset	Parameter Type & Value	Description
0	WORD = 4	Size of remaining parameter block
2	FARPTR	Address of font definition.

This function defines which character set is to be used by the normal text functions.

The font definition, whose address is required by this function, can be one of three different font types: monochrome bit-mapped, color bit-mapped, and short-stroke vector. For an example of how to use the latter with the regular text functions, refer to Chapter 5. For more detailed information about the font structure and definitions, refer to Appendix C.

Once this function has been called, the font-definition data area should not be changed in any way, because the AI may have extracted information that might no longer be valid. If you do need to change such data, call the HSCS function after such changes have been made.

Related functions: HCCHST, HCHST, and HXLATE.

HSGQ

Set graphics-quality information.

Byte Offset	Parameter Type & Value	Description
0	WORD = 2	Size of remaining parameter block
2	WORD	Graphics-quality flags

This function sets various internal AI flags that determine the "quality" of output. The definition of the flag's bits is as follows:

Bit(s)	Definition
15	Reserved
14	0 = high precision, 1 = low precision. Determines if the AI or adapter performs calculations such that objects are drawn correctly, to the nearest pixel position, or if objects are approximated and thus drawn faster. This operation is ignored in the 8514/A AI.
13	Reserved
12 – 11	00 = last pixel in a line is not drawn, 01 = last pixel in a line is drawn, 10 = last pixel in a line is conditionally drawn. In the last case, the issue of whether or not the last pixel in a line gets drawn is dependent on the mix mode. In the following mix

modes, the last pixel does get drawn (Dst refers to the pixel in display memory, and Src is the pixel intended to update display memory):

Dst = 0
Dst = 1 (all bits set)
Dst = Dst
Dst = NOT Src
Dst = Src
Dst = NOT Src OR Dst
Dst = Src OR Dst
Dst = Src AND Dst
Dst = NOT Src AND Dst
Dst = MINIMUM(Src, Dst)
Dst = MAXIMUM(Src, Dst)
Dst = Dst – Src [with saturate]

10	0 = automatically close areas to be filled by connecting the first point to the last via a line, 1 = don't close areas to be filled (up to the application)
9 – 0	Reserved

HSHS

Set the clipping rectangle.

Byte Offset	Parameter Type & Value	Description
0	WORD = 0 or 8	Size of remaining parameter block
2	WORD	Left limit (low X)
4	WORD	Right limit (high X)
6	WORD	Bottom limit (high Y)
8	WORD	Top limit (low Y)

This function sets the 8514/A hardware clipping rectangle. The purpose of the clipping rectangle is to prevent drawing commands from drawing outside a specific area.

If the parameter size field is 0, the default clipping rectangle is set. If the parameter field is 8, the specified values are used. If the left limit is greater than the right limit or the top limit is greater than the bottom limit, nothing will be drawn.

Pixels that lie on the clipping-rectangle boundary are drawn.

HSLPC

Save the line-pattern count.

Byte Offset	Parameter Type & Value	Description
0	WORD = 0	Size of remaining parameter block

No parameters are required.

This function saves the line-pattern count for later use by the HRLPC command.

The line-pattern count is a pointer into the line bit pattern. As pixels in a line are drawn, the index is incremented until the end of the pattern is reached, in which case it is reset to the beginning of the pattern.

Related function: HRLPC.

HSLT

Set the line-drawing type.

Byte Offset	Parameter Type & Value	Description
0	WORD = 1 or 6	Size of remaining parameter block
2	BYTE	Line-pattern type
4	FARPTR	Address of user-defined pattern

This function sets the line pattern to be used for drawing.

The line-pattern type is one of the following:

Value	Type & Pattern
0	Use user-defined pattern (see below)
1	Dotted line — 1 on, 2 off
2	Short dashed line — 5 on, 3 off
3	Dash-dot line — 6 on, 4 off, 2 on, 4 off
4	Double dotted line — 2 on, 4 off, 2 on, 8 off
5	Long dash line — 9 on, 3 off
6	Dash double-dot line — 8 on, 4 off, 2 on, 2 off, 2 on, 2 off
7	Solid line
8	Invisible line

The user-defined line-pattern data consists of a number of byte pairs, the first of which indicates an "on" count of pixels and the second of which is the "off" count. The byte pairs are preceded by a word that indicates the number of pairs following it. The 8514/A AI allows for user-defined patterns up to 48 pixels long. The format is as follows:

Byte Offset	Type	Description
0	WORD	Byte pair count (n)
2	BYTE	On count
3	BYTE	Off count
.		
.		
.		
n * 2 + 2	BYTE	On count
n * 2 + 3	BYTE	Off count

This function resets the line-pattern count.

Related functions: HRLPC and HSLPC.

HSLW

Set the line-drawing width.

Byte Offset	Parameter Type & Value	Description
0	WORD = 1	Size of remaining parameter block
2	BYTE	Line width

This function sets the line width to be used for line drawing.

The 8514/A AI supports only line widths of 1 or 3 pixels. If the width is specified as 0 or 1, a width of 1 is used. For a specified width greater than 1, a pixel width of 3 is used.

A line of width 3 is thickened by drawing an extra line to the left and right or above and below, depending on the angle of the line.

HSMARK

Define a marker.

Byte Offset	Parameter Type & Value	Description
0	WORD = 10 or 14	Size of remaining parameter block
2	BYTE	Marker width (in pixels)
3	BYTE	Marker height (in lines)
4	BYTE	Flags
5	BYTE	Reserved
6	WORD	Size of marker-definition data
8	FARPTR	Address of marker definition
12	FARPTR	Address of marker color data

This function defines the marker used by the marker drawing commands. A marker is a rectangular block of binary data. The definition of the marker starts with bit 7 of the first byte in the block and proceeds with sequential bytes until the entire marker is defined. There is no padding between the rows. The color data, if specified, consists of one byte containing a color value for each pixel in the marker definition, regardless of the bit pattern in the binary definition.

The flags are defined as follows:

Bit	Definition
7	0 = use monochrome definition, 1 = use color definition
6 – 0	Reserved

The size of the marker-definition data is calculated with the following formula:

size = ((width x height) + 7)/8

Related functions: HCMRK and HMRK.

HSMODE

Set a new mode.

Byte Offset	Parameter Type & Value	Description
0	WORD = 1	Size of remaining parameter block
2	WORD	Mode number

This function sets a new mode for the 8514/A to operate in. Valid mode numbers are:

Mode Number	Screen Size	Alpha Text Size	Text Resolution
0	1,024 x 768	12 by 20	85 by 38
1	640 x 480	8 by 14	80 by 34
2	1,024 x 768	8 by 14	128 by 54
3	1,024 x 768	7 by 15	146 by 52

All other modes are reserved for future use.

Related function: HOPEN.

HSMX

Set the mix modes.

Byte Offset	Parameter Type & Value	Description
0	WORD = 2	Size of remaining parameter block
2	BYTE	Foreground mix
4	BYTE	Background mix

This function sets the foreground and background mix. A mix (also known as a raster operation) is used to logically or mathematically combine the source pixel (from the active color, PC data, or display data) with the destination pixel (the pixel already at the destination in display memory). The result of the mix is placed in the destination-pixel position, pursuant to the setting of the plane enables, clipping rectangle, and color comparison. Valid mix values and their meanings are:

Mix Value	Description
0	Keep existing mix
1	Src OR Dst
2	Src
3	Reserved
4	Src XOR Dst
5	Dst
6	MAXIMUM(Src, Dst)
7	MINIMUM(Src, Dst)
8	Src + Dst [with saturation]
9	Dst – Src [with saturation]
10	Src – Dst [with saturation]
11	(Src + Dst)/2 [with overflow]
12	Reserved
13	Reserved
14	Reserved
15	Reserved
16	0, all bits cleared
17	Src AND Dst

18	Src AND NOT Dst
19	Src
20	NOT Src AND Dst
21	Dst
22	Src XOR Dst
23	Src OR Dst
24	NOT Src AND NOT Dst
25	NOT (Src XOR Dst)
26	NOT Dst
27	Src OR NOT Dst
28	NOT Src
29	NOT Src OR Dst
30	NOT Src OR NOT Dst
31	1, all bits set
32 – 255	Reserved

AND, OR, XOR, NOT are all bitwise binary operators. Saturate, as used above, means that the operations are bounded at 0 in the case of subtractions or at the maximum pixel value (15 or 255) in the case of additions.

HSPAL

Save the palette.

Byte Offset	Parameter Type & Value	Description
0	WORD = Special	Size of remaining parameter block
2	BYTE	Palette data

This functions saves the palette located on the board into the parameter block. The size of the palette save area is determined by a call to HQDPS. The palette is saved in a format appropriate for restoration using the HRPAL command.

Related function: HRPAL.

HSPATT

Define an area pattern.

Byte Offset	Parameter Type & Value	Description
0	WORD=10 or 14	Size of remaining parameter block
2	BYTE	Pattern width (in pixels)
3	BYTE	Pattern height (in lines)
4	BYTE	Flags
5	BYTE	Reserved
6	WORD	Size of pattern-definition data
8	FARPTR	Address of pattern definition
12	FARPTR	Address of pattern color data

This function defines the pattern used by the area-fill command. An area pattern is a rectangular block of binary data. The definition of the pattern starts with bit 7 of the first byte in the block and proceeds with sequential bytes until the entire pattern is defined. There is no padding between the rows. The color data, if specified, consists of one byte containing a color value for each pixel in the pattern definition, with the monochrome pattern determining which color bytes are actually used.

The area pattern also has a reference point specified by the HSPATTO command.

The flags are defined as follows:

Bit	Definition
7	1 = use monochrome definition, 0 = use color definition
6 – 0	Reserved

The size of the pattern-definition data is calculated by using the following formula:

size = ((width x height) + 7)/8

A solid-fill pattern is specified by using a width and height of 1. The maximum pattern size is 32 x 32.

Related function: HSPATTO, HBAR, HEAR.

HSPATTO

Set the pattern reference point.

Byte Offset	Parameter Type & Value	Description
0	WORD = 4	Size of remaining parameter block
2	WORD	X
4	WORD	Y

This function sets the pattern reference point for functions that use the pattern. The reference point refers to the upper left corner of the pattern and indicates where that corner is located on the display.

Related function: HSPATT.

HSYNC

Synchronize the AI with a task.

Byte Offset	Parameter Type & Value	Description
0	WORD = 2	Size of remaining parameter block
2	WORD	Task buffer segment

This function updates the task-dependent portion of the AI with the task information located at offset 0 of the segment specified in the parameter block. This command can be used to implement task switching.

Related function: HINIT.

HXLATE

Set a translation table for multicolor text font.

Byte Offset	Parameter Type & Value	Description
0	WORD = 32	Size of remaining parameter block
2	DWORD	8 color values

This function sets up a translation table for the multiplane bit-mapped fonts (see Appendix C) you draw with the HCCHST and HCHST functions. Multiplane fonts can consist of eight different colors, hence the eight color values in the parameter block.

5

Tutorial: The AI

Because it's difficult to determine how to use something like the AI properly without examples, we've provided you with this chapter, which gives several programming examples in C. We assume you have a good understanding of C, and so we don't cover any of the basics here. All the code shown here was developed using Microsoft C Version 5.1; other C compilers may require minor changes in the code. If you would like the code in magnetic form and didn't purchase a disk with this book, you can find information about how to obtain the disk in Appendix F.

In addition to the C compiler, you need a couple of include files, AFIDATA.H and IBMAFI.H, and an object file, CALLAFI.OBJ, which is linked with all AI code. All three files are supplied with the IBM 8514/A Display Adapter.

AFIDATA.H contains the data structures used to define the parameter blocks and such things as the font-file headers. We have provided our version of AFIDATA.H in Appendix E for your reference.

IBMAFI.H contains definitions for all the AI functions as well as a line that includes AFIDATA.H. This means that your C source files need only have the following line to be able to use the AI definitions:

```
#include <AFIDATA.H>
```

Make sure that both AFIDATA.H and IBMAFI.H are located in your include-file directory so that the C preprocessor can locate them.

Appendix D contains a listing of our version of this file, called IBMAFI.H.

CALLAFI.OBJ contains code that checks for the existence of the AI as well as a routine that is used to make calls to it. The source code, CALLAFI.ASM, is also supplied with the IBM 8514/A.

In the example code, we use a slightly different convention for declaring and casting variables. The definitions of our convention are in Listing 5-1. We have found this convention helps when communicating with assembly-language subroutines and working under other operating systems because we always know what size our parameters are (8, 16, or 32 bits). It also clarifies structure sizes.

Listing 5-1: Important definitions used in all the listings

```
/*
The following definitions are used in all the listings to reference
certain data types. This is done to help make the code more readable and
to assist later translation and communication with assembly language.
*/


#define  FALSE           0
#define  TRUE            !(FALSE)

#define  BYTE             char
#define  WORD           short
#define  DWORD         long
#define  UBYTE            unsigned BYTE
#define  UWORD          unsigned WORD
#define  UDWORD       unsigned DWORD
#define  FAR                 far
#define  HUGE              huge
#define  VOID               void
```

What's First?

The installation program supplied with the IBM 8514/A sets up directories on your disk with the AI files and fonts. There are two ver-

sions of the AI — one for the 4-bit-per-pixel board and one for the board with memory expansion, which supports 8 bits per pixel. The IBM install utility makes sure that the correct version of the AI is installed under the file name HDILOAD.EXE. You need to run this program prior to running any programs that use the AI.

The first thing your program needs to do when it wants to use the AI is to check that the AI is installed. You don't need to check for the presence of the graphics board because the AI will not allow itself to be installed if an appropriate board isn't present.

The following code snippet performs a check to see if the AI has been installed:

```
/*
   Sample C code to determine if AI is loaded and, if so,
   get the pointer to the jump table. The jump-table pointer
   is stored internal to the C binding.
*/

   if (!getafi())
      {
       printf("AI not loaded!\n");
      exit(1);
      }
```

Opening the Graphics Device

The first step in opening the AI is to use the HQDPS command to determine the sizes of the palette save area and the task state buffer. With this information, blocks of memory can be allocated to store the data.

Next, you open the AI for use by using the HOPEN command. HOPEN returns information that can be used to determine if any errors occurred while trying to open the AI.

The final step in getting the AI set up for general use is a call to the HINIT function, which loads the state information into the state buffer allocated earlier.

This opening sequence is presented as part of Listing 5-2.

Listing 5-2: Opening the AI — SKELETON.C

```
/*
      This program can be run by compiling it with the
      following line:

           CL -c LIST5-2.C
            LINK LIST5-2.C CALLAFI.OBJ

      Make sure to include a routine called DoSomething() in
      order to get a clean compile and see some results.

*/
/*--------------------------------------------------------------------------

    Include file section

    Some of these are required by the examples in Chapter 5.

------------------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <dos.h>
#include <conio.h>
#include <io.h>
#include <malloc.h>
#include <sys\types.h>
#include <sys\stat.h>
#include "ibmafi.h"                          /* Used for AI info              */


/*------------------------------------------------------------------------

    Code

------------------------------------------------------------------------*/
main(argc, argv)
```

```
WORD   argc;
BYTE   **argv;
{

/*
    Allocate the data areas for the HQDPS, HOPEN, and HINIT
    commands. The non-zero values for initialization are
    the parameter block sizes as defined in Chapter 5. The
    structure definitions for HQDPS_DATA, HOPEN_DATA, and
    HINIT_DATA are located in AFIDATA.H.
*/
static HQDPS_DATA         stateInfo = { 6, 0, 0, 0 };
static HOPEN_DATA         openData = { 3, 0, 0 };
static HINIT_DATA         stateData = { 2, 0 };

/*
    Allocate the data area for the HCLOSE command.
*/
static HCLOSE_DATA        closeData = { 2, 1 };
        BYTE               *stateBuffer;/* Pointer to task buffer */
        BYTE FAR           *statAddr;   /* Pointer to task buffer */

   if(!getafi())                /* Is AI loaded? */
      {
       printf("Error! No Adapter Interface Loaded.\n");
      exit(1);
      }

   /*
       Get the size of the state buffer. The C call "HQDPS" is
       actually a macro definition in IBMAFI.H. This applies
       to all the other AI calls as well.
   */
    HQDPS(&stateInfo);

   /*
       Use the state buffer size to dynamically allocate
       memory for the state buffer. The "+ 15" is necessary
       because HINIT requires the segment of the state buffer,
       and the buffer is assumed to start at offset 0 of that
       segment. By adding 15 to the size, we make sure that we
       have allocated enough space for the buffer even in the
       worst case situation in which the segment alignment is
       furthest off. Remember that segment alignment occurs at
       16 byte intervals where the linear address' least
       significant nibble is 0.
   */
    stateBuffer = (BYTE *) malloc(stateInfo.size + 15);
    statAddr = (BYTE FAR *) stateBuffer;

   /*
```

```
    Get the segment of the state buffer. Make sure to
    account for an allocation that is not segment boundary
    aligned.
*/
 stateData.segment = FP_SEG(statAddr) + ((FP_OFF(statAddr) + 15) >> 4);

/*
    Initialize the parameters for the HOPEN command and
    make the call. The HOPEN command opens the AI for use.
    After the call has returned, check the return flags for
    an error condition.
*/
 openData.oflags = 0x00;
 openData.mode = 0;     /* Set the 1024x768 mode */
 HOPEN(&openData);
 if (openData.iflags)
    {
    printf("An error occurred while trying to open the AI!\n");
    exit(2);
    }

/*
    No error occurred, so get the state information so we
    can start drawing.
*/
 HINIT(&stateData);

/*
    The following call has been included to indicate that
    this is the point at which you can insert your own
    graphics code, or many of the examples in this chapter.
*/
 DoSomething();

 getch();           /* Wait until the display has been viewed */

/*
    Now that we have done something, it's time to clean up
    after ourselves. This includes closing the AI and
    deallocating the state buffer.
*/
 HCLOSE(&closeData);
 free(stateBuffer);

 return(0);
}
```

As you may notice from the code, several references are made to macros and structures defined in the IBM-supplied files. To make things easier, we have provided a better documented version of the macros and structures in Listing 5-3.

Listing 5-3: The structures and macros used in Listing 5-2

```
/*
        The following macro definitions are used by the program in
Listing 5-2. The integer passed to callafi is the function number for
the given command. The X in the macro definition is a pointer to the
parameter block for the function being called.
*/
extern VOID_FAR callafi(WORD, BYTE FAR *);

#define HQDPS(X)    callafi(31, (BYTE FAR *) X)
#define HOPEN(X)    callafi( 8, (BYTE FAR *) X)
#define HINIT(X)    callafi(48, (BYTE FAR *) X)
#define HCLOSE(X)   callafi(34, (BYTE FAR *) X)

/*
        Here we have the four structures used in Listing 5-2. The
complete list and definition of AI structures is in Appendix E.
*/

typedef struct
        {
        UWORD   length;
        WORD    size;
        WORD    stack;
        WORD    palbufsize;
        } HQDPS_DATA;

typedef struct
        {
        UWORD   length;
        UBYTE   oflags;
        UBYTE   mode;
        UBYTE   iflags;
        } HOPEN_DATA;

typedef struct
        {
        UWORD   length;
        UWORD   segment;
        } HINIT_DATA;
```

```
typedef struct
        .
        UWORD   length;
        UBYTE   flags;
        | HCLOSE_DATA;
```

Note: In the code in this chapter, we have assumed that the AI and graphics board will properly support the 1,024 x 768 mode, but that is not necessarily a good assumption. If you want to make sure that your AI programs work at any current or new resolution, you should use the HQMODE command, after HOPEN, to determine the current resolution. Once you have the resolution, your program should adapt itself automatically.

Closing the Graphics Device

Closing the AI is substantially easier than opening it because only one call — namely, to HCLOSE — is required. The use of the HCLOSE command is shown at the end of Listing 5-2.

Let's Get Graphical

Now you know everything you need to know about setting up an AI program. All that's left is putting some things up on the screen. Actually, it's not that trivial, but it is still pretty straightforward. Before we start drawing, there are some things that need to be considered, such as color, mixes, color compares, plane masks, and clipping rectangles. Also not to be forgotten is the current drawing position.

Let's go through each one of these.

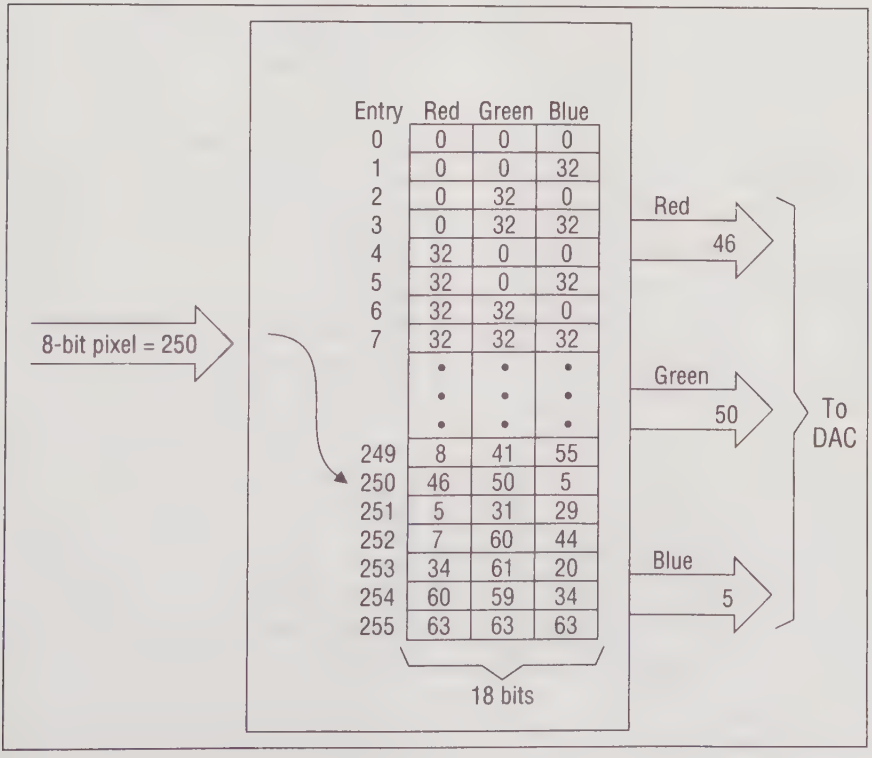
Color

The color of a pixel is dependent on two things: the digital value of the pixel in memory and the contents of the lookup table (LUT) for that value.

With the AI, the pixel's value is generally set using the HSCOL or HSBCOL command. These commands set the color for future drawing commands. In a 4-bit-per-pixel (bpp) configuration, the color value can range from 0 to 15; in an 8-bpp system, the values run from 0 to 255.

These color values are then run through the LUT and converted into three values: red, green, and blue. These three values control the intensities of the three respective color guns for a given pixel on the display. See Figure 5-1 for a diagram of this operation.

Figure 5-1: How pixel values are translated by the LUT



The default palette on the 8514/A has the first 16 entries set to the same IRGB combination used in color text modes, and the remaining 240 entries are set up in a similar pattern (see Table 5-1 for a list). The AI palette functions are HLDPAL, HRPAL, and HSPAL. Drawing colors are selected with HSCOL and HSBCOL, for foreground and background colors, respectively.

Table 5-1: Initial LUT definition

Entry	Color
0	Black
1	Dark blue
2	Dark green
3	Dark cyan
4	Dark red
5	Dark magenta
6	Dark yellow/brown
7	Gray
8	Dark gray (almost black)
9	Bright blue
10	Bright green
11	Bright cyan
12	Bright red
13	Bright magenta
14	Bright yellow
15	White
16 – 31	Dark blue
32 – 47	Dark green
48 – 63	Dark cyan
64 – 79	Dark red
80 – 95	Dark magenta
96 – 111	Dark yellow/brown
112 – 127	Gray
128 – 143	Dark gay (almost black)
144 – 159	Bright blue
160 – 175	Bright green
176 – 191	Bright cyan
192 – 207	Bright red
208– 223	Bright magenta
224 – 239	Bright yellow
240 – 255	White

This all boils down to the fact that you can assign a specific color to a pixel value. The color applies to all pixels of that value on the screen.

All other operations, such as HSMX, HSCMP, and HSBP, deal only with the value of a pixel, not its color.

Mixes

Mixes are used to manipulate the values of pixels before they are placed in display memory. A large number of operations can be performed on a pixel (see the explanation of the HSMX function in Chapter 4). When the AI is initialized (via HINIT), the foreground mix is set to replace/overpaint mode and the background mix is left the way it was.

Color Compares

The color-compare function built in to the 8514/A and supported by the AI allows for specific pixels in display memory to be excluded from updating. The protected pixels are selected by comparing them with a color-compare value. The color-compare takes precedence over the mix operation on a pixel. In other words, if the result of the color-compare operation is such that a pixel can be updated, then the result of the mix operation is placed in that pixel. The color-compare value is not set at initialization of the AI, but the color-compare function is set to FALSE to enable the update of all pixel values. See the HSCMP function in Chapter 4 for a more detailed explanation.

Plane Masks

Plane masks, specifically the write mask, also affect how pixels are updated in display memory. The write mask affects updates to display memory by controlling the ability to update the individual planes of a pixel (see the description of HSBP in Chapter 4). The write mask is set to all planes enabled upon AI initialization.

Clipping Rectangles

A clipping rectangle allows a program to limit the update of display memory to a specific rectangular subset of memory. This is useful in windowing environments because it ensures that drawing operations won't extend beyond the clipping rectangle (see HSHS in Chapter 4). When the AI is initialized, the clipping rectangle is set to the screen dimensions.

Current Position

As with many graphics interfaces, the AI supports both relative and absolute (x, y) addressing of display memory. Relative points are signed 8-bit values, relative to the current position. The current position is set directly by the HSCP command or indirectly by a number of drawing commands. It may also help to review the coordinate system used by the 8514/A and AI, described in Chapter 4. The current position is initialized to (0, 0).

For more information on how the AI initializes things, refer to the description of HINIT in Chapter 4.

Back to the Drawing Board

Now that we have covered the operation environment for drawing functions, lets look at some of these functions.

Line Drawing

Drawing lines is the first thing most people want to try on a graphics board. Lines are easy to draw using any of four AI commands: HLINE, HCLINE, HCRLINE, and HRLINE. The *C*s in the command names indicate that they use the current position as the first point in the line, and the *R*s signify that those commands support relative coordinates. All four of these commands are polyline commands, which means they can draw any number (less than about 32,000) of connected lines. Listing 5-4 shows four routines based on the four line commands that each draw a box at the point specified upon entry to the routine.

Listing 5-4: Four line-drawing examples

```
/*--------------------------------------------------------------------

    DrawBox1(x, y)

    Draw a 10x10 box with the upper left corner at x,y using HCLINE.
    Assume CP is already set to x,y. Use current drawing environment.

--------------------------------------------------------------------*/
VOID   DrawBox1(x, y)
WORD   x, y;
{
    HCLINE_DATA(4)    line;                  /* Define the data structure     */

    line.length = 16;                        /* 4 point pairs = 16 bytes      */

    line.coords[0].x_coord = x + 10;        /* Define the polyline           */
    line.coords[0].y_coord = y;             /* Upper right corner            */
    line.coords[1].x_coord = x + 10;
    line.coords[1].y_coord = y + 10;        /* Lower right corner            */
    line.coords[2].x_coord = x;
    line.coords[2].y_coord = y + 10;        /* Lower left corner             */
    line.coords[3].x_coord = x;
    line.coords[3].y_coord = y;              /* Back to upper left corner     */

    HCLINE(&line);                           /* Draw the box                  */

    return;
}
```

```
/*-------------------------------------------------------------------------

    DrawBox2(x, y)

    Draw a 10x10 box with the upper left corner at x,y using HCRLINE.
    Assume CP is already set to x,y. Use current drawing environment.

-------------------------------------------------------------------------*/
VOID   DrawBox2(x, y)
WORD   x, y;
{
    static HCRLINE_DATA(4) lineCR =
                    {
                         8,                  /* 4 relative point pairs      */
                        10,   0,             /* Upper right corner          */
                         0,  10,             /* Lower right                 */
                       -10,   0,             /* Lower left                  */
                         0, -10              /* Back to upper left          */
                    };

    HCRLINE(&lineCR);                        /* Draw the box                */

    return;
}

/*-------------------------------------------------------------------------

    DrawBox3(x, y)

    Draw a 10x10 box with the upper left corner at x,y using HLINE.
    Use current drawing environment.

------ ------------------------------------------------------------------*/
VOID   DrawBox3(x, y)
WORD   x, y;
{
    HLINE_DATA(5)  line;                     /* Define the data structure   */

    line.length = 20;                        /* 5 point pairs = 20 bytes    */

    line.coords[0].x_coord = x;              /* Define the polyline         */
    line.coords[0].y_coord = y;              /* Upper left corner           */
     line.coords[1].x_coord = x + 10;
    line.coords[1].y_coord = y;              /* Upper right corner          */
     line.coords[2].x_coord = x + 10;
    line.coords[2].y_coord = y + 10;         /* Lower right corner          */
     line.coords[3].x_coord = x;
    line.coords[3].y_coord = y + 10;         /* Lower left corner           */
     line.coords[4].x_coord = x;
    line.coords[4].y_coord = y;              /* Back to upper left corner   */
```

```
    HLINE(&line);                           /* Draw the box                 */

    return;
}

/*-----------------------------------------------------------------------------

    DrawBox4(x, y)

    Draw a 10x10 box with the upper left corner at x,y using HRLINE.
    Use current drawing environment.

-----------------------------------------------------------------------------*/
VOID   DrawBox4(x, y)
WORD   x, y;
{
    static HRLINE_DATA(4) lineR =
                    {
                        12,                 /* 4 relative point pairs + X,Y */
                         0,   0,            /* Starting position            */
                        10,   0,            /* Upper right corner           */
                         0,  10,            /* Lower right                  */
                       -10,   0,            /* Lower left                   */
                         0, -10             /* Back to upper left           */
                    };

    lineR.coord1.x_coord = x;
    lineR.coord1.y_coord = y;               /* Set the starting point       */

    HRLINE(&lineR);                         /* Draw the box                 */

    return;
}

/*-----------------------------------------------------------------------------

    DoSomething()

    Code called from the AI skeleton code in Listing 5-2 (LIST5-2.C)

    Draws boxes using the four box routines.

-----------------------------------------------------------------------------*/
VOID   DoSomething()
{
    HSCP_DATA    hscp;
    HSCOL_DATA   hscol;

    hscp.length = 4;                        /* 2 point pairs = 4 bytes      */

    hscol.length = 4;                       /* Color is 32 bits             */
```

```
    hscol.index = -1;                       /* Set color to all bits set,    */
    HSCOL(&hscol);                          /*  means white in normal palette*/

    hscp.coord1.x_coord = 50;
    hscp.coord1.y_coord = 50;               /* Initial box position
*/
    HSCP(&hscp);

    DrawBox1(50, 50);                       /* Box using HCLINE              */

    hscp.coord1.x_coord = 70;               /* Second box position           */
    HSCP(&hscp);

    DrawBox2(70, 50);                       /* Box using HCRLINE             */


    DrawBox3(50, 70);                       /* Box using HLINE               */

    DrawBox4(70, 70);                       /* Box using HRLINE              */

    return;
}
```

Note how the two relative commands can have their box-drawing information predefined. That's because of the way relative coordinates operate. Also, **DrawBox2**, which uses the HCRLINE function, doesn't actually require the start point as a parameter because it draws relative to the current point. Finally, **DrawBox1** could have been recoded to use the HQCP command to get the current position, instead of requiring the position as a parameter.

These routines demonstrate how dependent the AI is on the parameter blocks. If the same parameter block is going to be used again and again, it's best to define it as static or global and initialize as many fields as possible.

One thing you may notice in the AI command set is that there is no clear way to draw a single pixel. The easiest way to get around this problem is to use the line-drawing routines to draw a line one pixel long by specifying the same start and end points. It's not efficient, but it's the quickest way to do it with the AI.

The line commands also have a few line-specific modifiers, such as line pattern and line width.

The line width can be set (with the HSLW command) to only one or three pixels in the 8514/A version of the AI, and the three-pixel-wide lines are not quite accurate because they are drawn as three separate lines set one pixel apart. This introduces errors when certain types of mixes are used because adjacent lines usually overlap somewhat because of the pixel-oriented nature of the line drawing.

Setting the line pattern is less flaky and more useful. The AI lets you select eight predefined line patterns, in addition to a user-defined pattern. You select these patterns with the HSLT command. As a line is drawn, a counter in the AI determines where in the pattern to read from for the next pattern information. When the counter reaches the end of the pattern, it cycles back to the beginning. This is important because some of the line-drawing operations — namely, HCRLINE and HRLINE — reset the counter to the start of the pattern, whereas others continue to use the counter from the previous operation. To optionally preserve the counter, the AI has two functions — HRLPC and HSLPC — which restore and save the counter, respectively. You may want to do this when highlighting a line-based object with "marching ants" — a line pattern that is continually shifting by one position.

Markers

A graphical function that many graphics interfaces implement is drawing markers. A marker is a graphical symbol that can be placed at any point. The symbol is defined with either a single-plane bit map or a color bit map that can be up 255 pixels square. The function used to define the symbol is HSMARK. The marks are then BitBLTed onto the display with HCMRK and HMRK. Both of these are actually "poly"-mark functions, and therefore multiple markers can be drawn with just one call.

Listing 5-5 shows how to use the marker commands. In the listing, we show the use of monochrome bit maps only. Note that the most significant bit of the first byte is the first bit to be expanded and displayed. Color markers are left as an exercise.

Listing 5-5: Use of markers

```
/*-----------------------------------------------------------------------

    DoSomething()

    Code called from the AI skeleton code in Listing 5-2 (LIST5-2.C)

    Defines a monochrome mark and draws it at points along a formula.

-----------------------------------------------------------------------*/
VOID   DoSomething()
{
    static BYTE markBitmap[14][2] =
                  {
                   0xFF, 0xFF, /* 1111111111111111 */
                   0xC0, 0x03, /* 1100000000000011 */
                   0xA0, 0x05, /* 1010000000000101 */
                   0x90, 0x09, /* 1001000000001001 */
                   0x88, 0x11, /* 1000100000010001 */
                   0x87, 0xE1, /* 1000011111100001 */
                   0x84, 0x21, /* 1000010000100001 */
                   0x84, 0x21, /* 1000010000100001 */
                   0x87, 0xE1, /* 1000011111100001 */
                   0x88, 0x11, /* 1000100000010001 */
                   0x90, 0x09, /* 1001000000001001 */
                   0xA0, 0x05, /* 1010000000000101 */
                   0xC0, 0x03, /* 1100000000000011 */
                   0xFF, 0xFF  /* 1111111111111111 */
                  };

    HSMARK_DATA     hsMark;
    HMRK_DATA(13)   marks;
    HSCOL_DATA      hscol;
    WORD            i, j;

    hscol.length = 4;                       /* Color is 32 bits          */
    hscol.index =  14;                      /* Set color to yellow       */
    HSCOL(&hscol);                          /*  (in normal palette)      */


    hsMark.length = 10;                     /* Size for monochrome mark  */
```

```
    hsMark.width = 16;                          /* Width of the mark (in pixels) */
    hsMark.height = 14;                         /* Height in lines               */
    hsMark.flags = 0;                           /* Monochrome mark               */
    hsMark.res = 0;                             /* Reserved words are always 0   */
    hsMark.imlen = sizeof(markBitmap);          /* Length of mark data in bytes  */
    hsMark.image = (UBYTE FAR *)markBitmap;/* Pointer to mark data               */
    HSMARK(&hsMark);

    marks.length = sizeof(marks) - 2;           /* Simple way to set length      */

    j = 6;
    i = 13;                                     /* Initialize some loop vars     */

    while (i--)                                 /* Plot a facsimile of x**3      */
        {
        marks.coords[i].x_coord = (j + 20) * 20;
        marks.coords[i].y_coord = j * j * j + 250;
        j--;
        }

    HMRK(&marks);

    return;
}
```

Rectangles

The 8514/A excels at a few drawing functions, and filled rectangles are among them. These rectangles can be filled either solidly or with a pattern. The AI command for drawing filled rectangles is HRECT. Area patterns are specified with the HSPATT and HSPATTO commands. Listing 5-6 gives an example of how to fill solid and patterned rectangles. Note that to pattern fill a rectangle, we actually have to cheat by using HBAR and HEAR to define the region to be filled by the pattern, since patterns really only apply to areas and not rectangles.

One thing you can use filled rectangles for is clearing the screen or portions of it. To clear the entire display, you may want to use the HEGS command. HEGS clears the display to 0, but just like HRECT, it has the drawback of being affected by the clipping rectangle and plane enable.

Listing 5-6: Example of filled rectangles with and without patterns

```
/*---- -------------   ----- -------  ------- ------------------------------

    DoSomething()

    Code called from the AI skeleton code in Listing 5-2 (LIST5-2.C)

    Draws two type of rectangles, one solid filled and one pattern filled.
    Note that the pattern actually only applies to area fills, so we
    cheat by outlining all the rectangles and then doing a rectangle fill
    on-them.

--------------------- ----    ------------------------------------------*/
VOID   DoSomething()
{
                                                /* Psuedo-anti-aliased diamond   */
    static BYTE monoPattern[14] =
                    {
                     0x10,    /* 00010000 */
                     0x38,    /* 00111000 */
                     0x38,    /* 00111000 */
                     0x7C,    /* 01111100 */
                     0x7C,    /* 01111100 */
                     0xFE,    /* 11111110 */
                     0xFF,    /* 11111111 */
                     0xFE,    /* 11111110 */
                     0x7C,    /* 01111100 */
                     0x7C,    /* 01111100 */
                     0x38,    /* 00111000 */
                     0x38,    /* 00111000 */
                     0x10,    /* 00010000 */
                     0x10     /* 00010000 */
                    };
                                                /* Psuedo-anti-aliased diamond   */
    static BYTE colorPattern[14][8] =
                    {
                     0, 0, 0, 9, 0, 0, 0, 0,  /* ...B.... */
                     0, 0, 1, 9, 1, 0, 0, 0,  /* ..bBb... */
                     0, 0, 9, 9, 9, 0, 0, 0,  /* ..BBB... */
                     0, 1, 9, 9, 9, 1, 0, 0,  /* .bBBBb.. */
                     0, 9, 9, 9, 9, 9, 0, 0,  /* .BBBBB.. */
                     1, 9, 9, 9, 9, 9, 1, 0,  /* bBBBBBb. */
                     9, 9, 9, 9, 9, 9, 9, 1,  /* BBBBBBBb */
                     1, 9, 9, 9, 9, 9, 1, 0,  /* bBBBBBb. */
                     0, 9, 9, 9, 9, 9, 0, 0,  /* .BBBBB.. */
                     0, 1, 9, 9, 9, 1, 0, 0,  /* .bBBBb.. */
                     0, 0, 9, 9, 9, 0, 0, 0,  /* ..BBB... */
                     0, 0, 1, 9, 1, 0, 0, 0,  /* ..bBb... */
                     0, 0, 0, 9, 0, 0, 0, 0,  /* ...B.... */
```

```
                0, 0, 0, 1, 0, 0, 0, 0    /* ...b.... */
               };

static HRLINE_DATA(4) lineR =
               {
                 12,                      /* 4 relative point pairs + X,Y  */
                  0,   0,                 /* Starting position             */
                 48,   0,                 /* Upper right corner            */
                  0,  42,                 /* Lower right                   */
                -48,   0,                 /* Lower left                    */
                  0, -42                  /* Back to upper left            */
               };

HRECT_DATA      rect;
HSCOL_DATA      hscol;
HSPATT_DATA     pattInfo;
HSPATTO_DATA    pattRef;
HEAR_DATA       hearData;
WORD            i, j;

hscol.length = 4;                        /* Color is 32 bits              */
hscol.index =  9;                        /* Set color to blue             */
HSCOL(&hscol);                           /*  (in normal palette)          */

rect.length = 8;
rect.width = 48;
rect.height = 42;                         /* Set fixed parts of rectangle  */

for (i = 0; i < 20; i++)                  /* Draw solid based checkerboard */
    for (j = 0; j < 17; j++)
       if (!((i + j) & 1))
          {
           rect.coord.x_coord = i * 48;
           rect.coord.y_coord = j * 42;
          HRECT(&rect);
          }

pattInfo.length = 14;                    /* Size for color pattern        */
pattInfo.width = 8;                       /* Width of the pattern (in pix) */
pattInfo.height = 14;                    /* Height in lines               */
pattInfo.flags = 0;                      /* Color patttern                */
pattInfo.reserved = 0;                    /* Reserved words are always 0   */
pattInfo.img_length = sizeof(colorPattern);    /* Size of pattern         */
pattInfo.image = (UBYTE FAR *)monoPattern;     /* Pointer to mono pattern*/
pattInfo.colour = (UBYTE FAR *)colorPattern;   /* Pointer to color pattrn*/

HSPATT(&pattInfo);

pattRef.length = 4;
pattRef.coord1.x_coord = 0;               /* Reference is upper left corner*/
pattRef.coord1.y_coord = 0;
```

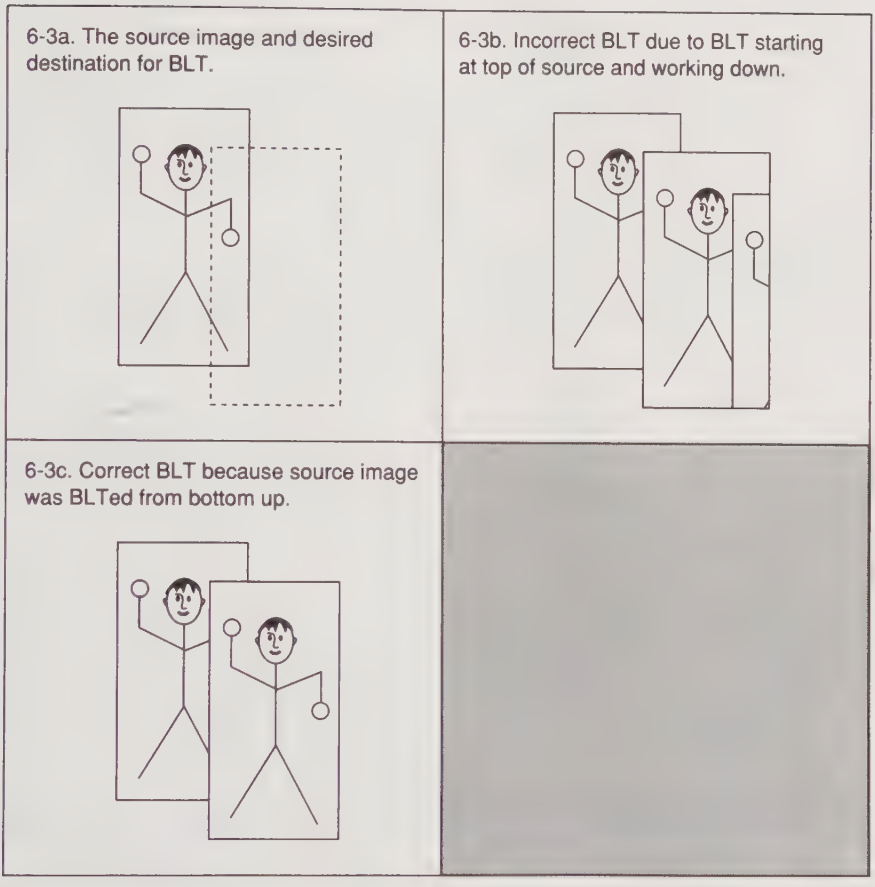
```
    HSPATTO(&pattRef);

    HBAR();                                    /* Start area fill definition    */
    for (i = 0; i < 20; i++)                   /* Used patterns in blank spots  */
       for (j = 0; j < 17; j++)
          if ((i + j) & 1)
             {
              lineR.coord1.x_coord = i * 48;
              lineR.coord1.y_coord = j * 42;/* Set the starting point          */
             HRLINE(&lineR);                   /* Draw the box                  */
             }

    hearData.length = 1;
    hearData.flags = 0;                        /* Fill the area                 */
    HEAR(&hearData);

    return;
}
```

BitBLTs

Bit-BLock Transfers (BitBLTs) are used to copy rectangular blocks of graphical data from one position to another. A proper BitBLT can handle overlap of the source and destination areas such that the source area will always be copied in full. Figure 5-2 shows some examples of how BLTs should and shouldn't work. The AI handles this properly, but as you may discover, you have to account for the overlap when programming the registers directly.

Two types of BLT operations are possible — all-plane copy and single-plane expansion. An all-plane-copy BitBLT is what we would consider a normal BLT — all pixels from the source region are copied to the destination, pursuant to the various mixes, clipping rectangles, and so on.

Figure 5-2: How BLTs should and shouldn't work


6-3a. The source image and desired destination for BLT.
6-3b. Incorrect BLT due to BLT starting at top of source and working down.
6-3c. Correct BLT because source image was BLTed from bottom up.

A single-plane expansion is generally used to take monochrome (single plane) data and expand it into the foreground color for bits that are set and the background color for bits that are cleared. This capability is sometimes used to display bit-mapped fonts. With the AI, you must specify the source plane to use as the single-plane source when doing the expansion BLT.

Listing 5-7 has examples of both types of BitBLTs using the AI HBBC command.

Listing 5-7: Example use of HBBC

```
/*  . .. .... ...  ... .... .... .......... ............ ..................

    DoSomething()

    Code called from the AI skeleton code in Listing 5-2 (LIST5-2.C)

    Demonstrates the use of all plane and single plane BitBLTs.

-------------------------------------------------------------------------*/
VOID DoSomething()
{
    static HRLINE_DATA(7)  lines =          /* Define the data structure     */
                        {
                        18,                 /* 7 Points + X,Y               */
                        0, 400,             /* Initial point                */
                          9,  0,  0,  9,    /* Top and right of box         */
                         -9,  0,  0, -9,    /* Bottom and left of box       */
                          9,  9,  0, -9,    /* First part of X, back to top */
                         -9,  9             /* Second part of X             */
                        };

    HBBC_DATA       bltInfo;
    HRECT_DATA      rect;
    HSCOL_DATA      hscol;
    WORD            i, j;

    hscol.length = 4;                       /* Color is 32 bits             */

    rect.length = 8;
    rect.width = 100;
    rect.height = 100;                       /* Set fixed parts of rectangle */

    /*
        Set up something that we can do an all plane BLT on. In this case,
        it's a 4x4 block of different colored rectangles.
    */
    for (i = 0; i < 4; i++)
       for (j = 0; j < 4; j++)
          {
          hscol.index =  j * 4 + i;         /* Set color                    */
          HSCOL(&hscol);
          rect.coord.x_coord = i * 100;
          rect.coord.y_coord = j * 100;
          HRECT(&rect);                     /* Draw the rectangle           */
          }

    bltInfo.length = 16;                     /* Size of HBBC parameter block */
    bltInfo.format = 8;                     /* All plane copy               */
```

```
    bltInfo.width = 400;                    /* Size of block                */
    bltInfo.height = 400;
    bltInfo.plane = 0;                      /* Plane is irrelevant in all pln*/
    bltInfo.reserved = 0;                   /* Golden rule                  */
    bltInfo.source.x_coord = 0;             /* Source block position        */
    bltInfo.source.y_coord = 0;
    bltInfo.dest.x_coord = 500;             /* Destination of BLT           */
    bltInfo.dest.y_coord = 100;
    HBBC(&bltInfo);                         /* Do the BitBLT                */

    /*
        Now we need to demonstrate the single plane expansion. Let's do so
        by simulating a single plane source. We'll draw a box with an X in
        it in plane 2, and then do a BLT expansion of the object into a
        array of multi-colored versions of the object.
    */

    hscol.index =  4;                       /* Set color for drawing plane 2 */
    HSCOL(&hscol);

    HRLINE(&lines);                         /* Draw the object.             */

    bltInfo.format = 0;                     /* Single plane expansion       */
    bltInfo.width = 10;                     /* Size of object               */
    bltInfo.height = 10;                    /* A delta of 9 is really 10 pts */
    bltInfo.plane = 2;                      /* Plane is #2                  */
    bltInfo.source.x_coord = 0;             /* Source block position        */
    bltInfo.source.y_coord = 400;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            {
            hscol.index =  i;               /* Set foreground color         */
            HSCOL(&hscol);
            hscol.index =  j;               /* Set background color         */
            HSBCOL(&hscol);
            bltInfo.dest.x_coord = i * 10 + 100; /* Destination of BLT      */
            bltInfo.dest.y_coord = j * 10 + 400;
            HBBC(&bltInfo);                 /* Do the BitBLT                */
            }

    return;
}
```

Data Transfer

An extension of the 8514/A BitBLT function is the ability to transfer data to and from the 8514/A. The AI supports this functionality via HBBW, HCBBW, and HBBCHN for writing data to the board and via HBBR and HBBCHN for reading data from board. As with normal BitBLTs, these data-transfer operations support the all-plane copy and single-plane-expansion modes.

You may notice that HBBCHN is used for both reading and writing data. This is because these transfer operations are two-step operations. The first step is the setup of the operation — this is when HBBR, HBBW, or HCBBW is used. The second step is the use of the HBBCHN function, which actually does the data transfer. HBBCHN is used as often as needed to transfer all the specified data. This is the only way to transfer bit-map information to or from the board.

An example of both reading and writing board data in the all-plane mode is given in Listing 5-8. The single-plane version of this code is left as an exercise. A derivative of the code in Listing 5-8 can be used to load image files to the board.

Listing 5-8: Example of data transfer between PC and board

```
/*-----------------------------------------------------------------------

    DoSomething()

    Code called from the AI skeleton code in Listing 5-2 (LIST5-2.C)

    Demonstrates the use of all plane read and write transfers.

-----------------------------------------------------------------------*/
VOID   DoSomething()
{
    HBBR_DATA       readInfo;
    HBBW_DATA       writeInfo;
    HBBCHN_DATA     chainInfo;
    HRECT_DATA      rect;
    HSCOL_DATA      hscol;
```

```
WORD            i, j;
UBYTE           *tmpBuffer;

hscol.length = 4;                       /* Color is 32 bits              */

rect.length = 8;
rect.width = 10;
rect.height = 10;                       /* Set fixed parts of rectangle  */

/*
    Set up something that we can read from the board. In this case,
    it's a 4x4 block of different colored rectangles.
*/
for (i = 0; i < 4; i++)
   for (j = 0; j < 4; j++)
      {
      hscol.index =  j * 4 + i;         /* Set color                     */
      HSCOL(&hscol);
      rect.coord.x_coord = i * 10;
      rect.coord.y_coord = j * 10;
      HRECT(&rect);                     /* Draw the rectangle            */
      }

tmpBuffer = (BYTE *)malloc(40 * 40);    /* Allocate memory for storage   */

/*
    First we need to set up the read operation. This tells the AI and board
    how much data and what kind of data we want to read, as well as where
    we want to read it from. It is an option to read the data into a rect-
    angluar region in PC memory. For this example, we are just doing a
    simple transfer.
*/
readInfo.length = 12;                   /* Size of HBBR parameter block  */
                                        /* (Also means simple transfer)  */
readInfo.format = 8;                    /* All plane copy                */
readInfo.width = 40;                    /* Size of block                 */
readInfo.height = 40;
readInfo.plane = 0;                     /* Plane is irrelevant in all pln*/
readInfo.reserved = 0;                  /* Golden rule                   */
readInfo.coord.x_coord = 0;             /* Source block position         */
readInfo.coord.y_coord = 0;
HBBR(&readInfo);                        /* Set up the Read operation     */

/*
    Next, we need to read the data from the board via the HBBCHN command.
    We'll do this one line at a time. A line is 40 pixels, ergo 40 bytes.
*/

chainInfo.length = 6;
chainInfo.len = 40;                     /* Non-changing info.            */
```

```
    for (i = 0; i < 40; i++)                    /* Read all forty lines.          */
       {
        chainInfo.address = (UBYTE FAR *)&tmpBuffer[i * 40];
        HBBCHN(&chainInfo);
       }

    /*
        Now let's write the same back up to the board, but at a different
        location. Again, we want to start with a command to tell the AI
        that we want to start a write transfer. Data is referenced as
        linear data, not rectangular, on the PC side. (We could have use
        the "readInfo" structure here instead of the "writeInfo", since
        they are identical.)

        For a better effect, let's clear the screen first.
    */
    HEGS();


    writeInfo.length = 10;                       /* Size of HBBW parameter block  */
                                                 /* (Also means simple transfer)  */
    writeInfo.format = 8;                       /* All plane copy                 */
    writeInfo.width = 40;                       /* Size of block                  */
    writeInfo.height = 40;
    writeInfo.coord.x_coord = 200;              /* Source block position          */
    writeInfo.coord.y_coord = 200;
    HBBW(&writeInfo);                            /* Set up the write operation    */

    /*
        Next, we need to write the data to the board via the HBBCHN command.
        This time we'll transfer it all at once.
    */

    chainInfo.len = 40 * 40;                    /* Buffer size.                   */
    chainInfo.address = (UBYTE FAR *)tmpBuffer;
    HBBCHN(&chainInfo);                         /* Write it all.                  */

    return;
}
```

Text

The AI supports text through three different font types: short stroke, monochrome bit-mapped, and multiplane bit-mapped. Text can be displayed via the AI using one of two methods.

The first is with the normal text commands: HSCS, which is used to tell the AI about a loaded font; and HCCHST and HCHST, which draw text strings. The normal text commands allow you to position text at any pixel location. All text colors are controlled by the standard means, except for multiplane bit-mapped fonts, which have their color specified with the HXLATE command.

The second method for displaying text involves the AI commands that start with *A*. These are the alphanumeric commands. They are based on a specific font cell size, determined by the current mode (see HOPEN, HSMODE, and ASCELL). The screen is addressed as a series of character rows and columns, using the cell size as a measure. This limits the positioning of the text characters, but on the other hand, you can specify the colors and text attributes for each character displayed without having to use separate AI commands. This approach is somewhat similar to the way the standard CGA, EGA, and VGA text modes work.

We concentrate on the normal text commands in our example, which is in Listing 5-9. The font-loading routine is the same one you would use for loading the font for alphanumeric operations. This is because the fonts used for both display methods are identical, and as such, they can be loaded in the same fashion.

Listing 5-9: Using fonts via the AI text routines

```
static    FFDEF        *ioAddr;              /* Address to load font to       */

/*        ...          ...                   ...                           ...

    LoadFont(name)

    This routine loads fonts from the files supplied by IBM with the 8514/A.
    The function requires the file name of the AI compatible font file, and
    if the font is successfully loaded, returns a pointer to the character
    set definition, as required by the AI. If the return value is 0, an
    error occurred while trying to load the file. Also, if you really want
    to be neat about things, the memory allocated for the font should be
    freed prior to exiting to DOS. Refer to Appendix ? for more information
```

```
    on font files.

    Note: FFDEF and CHARSETDEF are defined in our version of AFIDATA.H but
          not in IBM's.

-- ------ ---- -- ---- ------------------ ---- ---- ------- ------------*/
CHARSETDEF FAR   *LoadFont(name)
BYTE   *name;
{
          UWORD        fontLen;              /* Size of font to load          */
          WORD         fontHdl;              /* Font file handle              */
          CHARSETDEF   *pCSD;                /* Pointer to Char Set Def.      */

                                             /* Open the Font file            */
    if ((fontHdl = open(name, O_RDONLY|O_BINARY)) == -1)
       {
        printf("Could not open font file '%s'!!!\n", name);
       return(0L);
       }
                                             /* Get size of font file         */
    fontLen = (UWORD) lseek(fontHdl, 0L, SEEK_END);

    ioAddr = (FFDEF *) malloc(fontLen);      /* Allocate enough space for file*/
    if (!ioAddr)
       {
        printf("Not enough memory to load font!\n");
       return(0L);
       }

    lseek(fontHdl, 0L, SEEK_SET);            /* Return to start of file       */
    read(fontHdl, (BYTE *)ioAddr, fontLen);/* Load the whole file into mem. */

                                              /* Point to start of def. font   */
    pCSD = (CHARSETDEF *) (((BYTE *) ioAddr) +
               ioAddr->page_array[ioAddr->def_page].csd_offset);

                              /* Update all address fields with load address */
     pCSD->chardef1 = ((UBYTE FAR *) ioAddr) + ((DWORD) pCSD->chardef1);
     pCSD->chardef2 = ((UBYTE FAR *) ioAddr) + ((DWORD) pCSD->chardef2);
     pCSD->chardef3 = ((UBYTE FAR *) ioAddr) + ((DWORD) pCSD->chardef3);
     pCSD->indextbl = (WORD FAR *)
                         (((UBYTE FAR *) ioAddr) + ((DWORD) pCSD->indextbl));
     pCSD->enveltbl = ((UBYTE FAR *) ioAddr) + ((DWORD) pCSD->enveltbl);


    close(fontHdl);                          /* Close the font handle         */

    return(pCSD);                             /* Return pointer to the font    */
}
```

```
/*-------------------------------------------------------------------------

    DoSomething()

    Code called from the AI skeleton code in Listing 5-2 (LIST5-2.C)

    Demonstrates the use of the normal text commands.

-------------------------------------------------------------------------*/
VOID   DoSomething()
{
   HSCS_DATA          setChar;
   HCHST_DATA(80)     strInfo;
   HSCOL_DATA         hscol;

   hscol.length = 4;                        /* Color is 32 bits            */

   setChar.length = 4;                      /* Load the font - makes an    */
                                            /*  assumption about file name */
    setChar.address = LoadFont("\\hdipcdos\\stan1220.fnt");
    if (!setChar.address)
       return;                              /* If an error occurred, return */
   HSCS(&setChar);                          /* Set the character set       */

   hscol.index = -1;                        /* Default to white color      */
   HSCOL(&hscol);

    strInfo.coord.x_coord = 200;
   strInfo.coord.y_coord = 100;             /* Let's put it some place     */
   strcpy(strInfo.string, "Hello World!");      /* Sample string :-)       */
    strInfo.length = strlen(strInfo.string) + 4; /* Sneaky way to get length*/
   HCHST(&strInfo);                         /* Display the string.         */

   /*
       NOTE: Short stroke fonts, which the STAN????.FNT files happen to
       be, do not clear the background when displayed. This means that
       you need to do a filled rectangle to clear an area before displaying
       short stroke font characters. This can have some positive uses as well,
       especially if you don't want to alter the background of the character
      cell.
   */
   free(ioAddr);                            /* Free up the font buffer.    */
   return;
}
```

Summary

This chapter provides you with an overview of how to get started using the AI in your graphics programs. If you would like additional information, we recommend you purchase the two IBM documents referred to at the beginning of Chapter 4. The disk that comes with the programmer's guide contains several useful utilities that may aid you in debugging your AI programs.

If you are interested in using assembly language with the AI, refer to GETAFI.ASM, shipped with the IBM 8514/A Display Adapter.

6

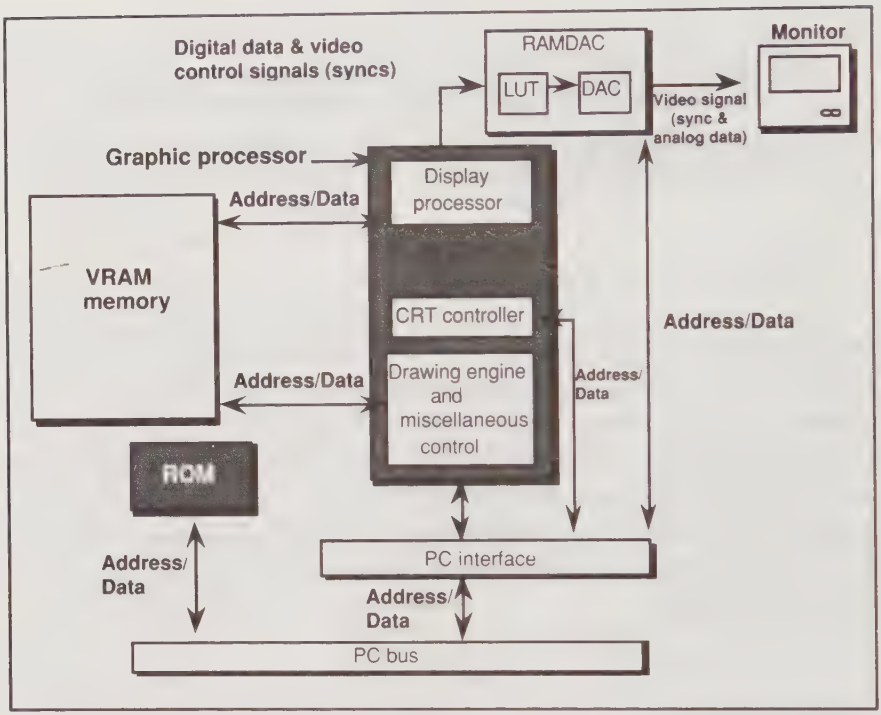
The 8514/A Hardware Interface and Register Set

In the previous two chapters, we looked at how you use an 8514/A via the Adapter Interface. Although using the AI will, according to IBM, guarantee compatibility with future IBM graphics platforms, if you need to squeeze as much performance out of the 8514/A as possible, going directly to the hardware is the best way to do so. Our informal benchmarks have shown a two-times increase in performance when going to the 8514/A registers instead of through the AI for graphics-intensive programs.

Architecture

The core of the IBM 8514/A consists of two chips — one to handle pixel manipulations and the other to deal with display generation. From a software perspective, there is no way to discern which chip is being accessed, nor does it really matter. As a matter of fact, several 8514/A-clone companies have taken the functionality of those two chips and integrated them into one chip, and in some cases they have included VGA functionality in that single chip as well. A block diagram of the 8514/A is shown in Figure 6-1.

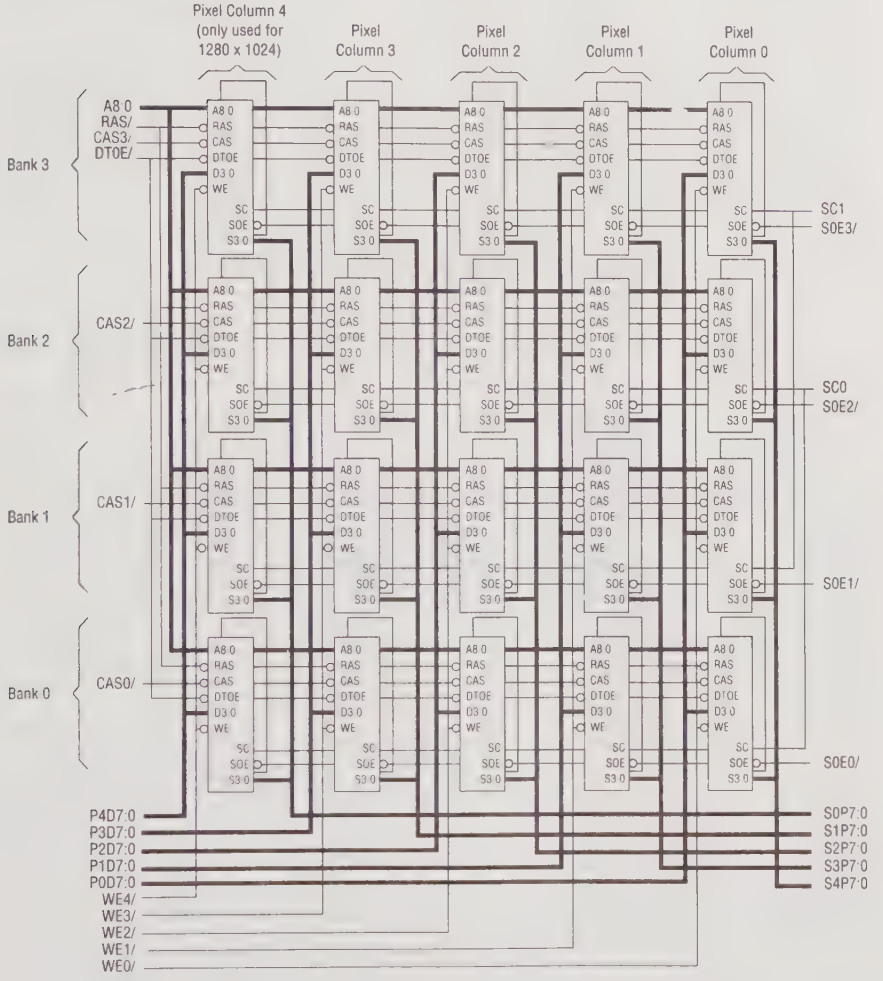
6-1: Block diagram of the 8514/A



Memory

The memory architecture of the 8514/A bears examination because it explains some of the oddities you find when reading through the register information. The IBM 8514/A uses video RAM (VRAM) for its display memory. VRAM allows the 8514/A to draw and display graphics simultaneously. On a normal DRAM graphics board, such as an EGA, wait states are introduced during drawing operations because display and drawing can't occur at the same time. New DRAM technology is helping minimize this problem, though, as evidenced by the DRAM-based 8514/A clone chips on the market.

Figure 6-2: IBM 8514/A memory organization using 64K x 4 VRAMs

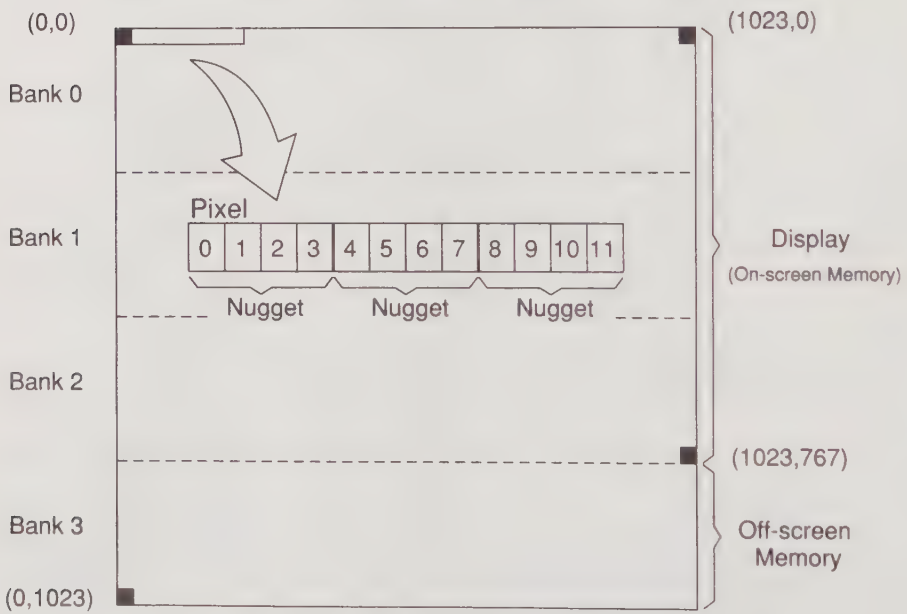


Notes: 1. Gray VRAMs are used only for 8 plane systems to hold upper nibble of pixel
2. Pixel Column 4 is used only in 5-pixel nugget systems, namely those designed for a horizontal resolution of 1280 pixels

(Courtesy of Chips and Technologies, San Jose, CA)

The IBM 8514/A uses 64K x 4 VRAMs, which means that each VRAM chip can hold 256K bits of information. The base 512K configuration uses 16 VRAM chips, arranged in a 4 x 4 array (see Figure 6-2). Each of the four banks represents 256 rows of 1,024 pixels. Within each bank, each of the four VRAM chips represents a 4-bit-per-pixel column (see Figure 6-3). Chips and Technologies has referred to the 4-pixel block as a *nugget*, and we'll use that term too, for lack of a better one.

Figure 6-3: Relation between memory architecture and contents



When the 512K expansion module is added to the base IBM 8514/A, another 4 bits of information are added to each pixel, resulting in 8-bit pixels. Note that a resolution of 1,280 x 1,024 can be achieved by adding a fifth VRAM to each bank, thereby increasing the nugget size to 5 pixels. Of course, the IBM 8514/A doesn't have sockets for the extra RAM, but some of the clone boards do.

One inherent advantage of this type of psuedoplanar architecture is that writing 8-bit pixels takes as much time as writing 4-bit pixels, whereas on a normal VRAM-based graphics board, such as one based on the TI 34010, 8-bit pixels can take twice as much time to write as an equal number of 4-bit pixels. The 8514/A memory architecture can also be utilized to produce graphics boards with more than than 8 bits per pixel, again with no noticeable performance degradation. Several clone companies have indicated that by combining multiple 8514/A chip sets, you can add more color depth to your pixels.

PC Interface

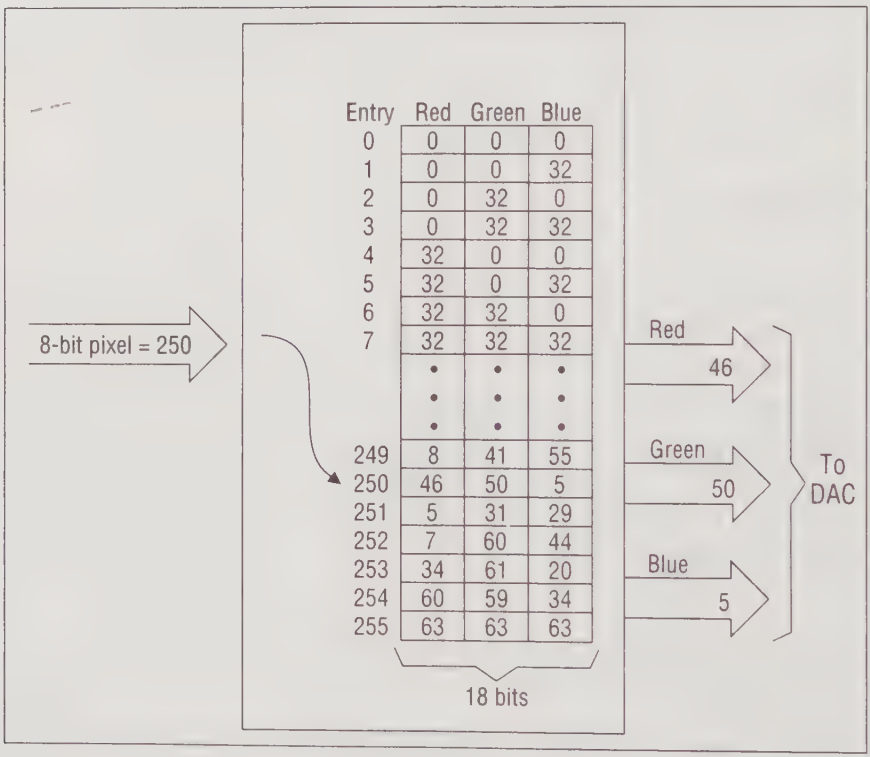
The PC-interface portion of the 8514/A handles all data transfers to and from the 8514/A graphics adapter. On the 8514/A, all communications are handled by reading or writing I/O ports. The IBM implementation of the board does not support any direct memory access, although some clone chips can support this capability. Note that most of the I/O operations on the 8514/A registers are word (16-bit) operations. A list of the registers that can be accessed is presented in Table 6-1 and in Appendix F.

Lookup Table (LUT)

The LUT allows a numeric pixel value to be translated into three color values — red, green, and blue — which are used to set the gun intensity of a color pixel on an analog RGB monitor. The LUT on the IBM 8514/A board is an 18-bit LUT, which means that each of the three color signals (red, green, and blue) can have up to 6-bit intensities (64 levels), for a total of 262,144 possible colors. Of course, only 256 colors can be displayed simultaneously in a fully configured 8514/A, but these colors can be chosen from the full palette of 262,144. The translation process used in the LUT is shown in Figure 6-4. Note that the LUT is often referred to as a RAMDAC, or just DAC (digital-

to-analog converter), because the most popular LUT chips also have built-in DACs.

Figure 6-4: Translation process used in the LUT



VGA Loop-Through

Also known as the VGA pass-through, this is basically a software-driven switch that determines whether the output of an attached VGA or the output of the 8514/A is visible on the display attached to the board. On the PS/2 Micro Channel platform, the only one on which IBM's board is available, the VGA-signal information is

passed to the 8514/A board via a video connector on one of the Micro Channel slots, thereby providing the 8514/A with full backward compatibility as well as permitting the use of a single-screen system.

The one catch is that these VGA signals are the digital information that is available prior to translation through the VGA's LUT, which means the 8514/A's LUT has to be used to generate the proper colors for the display if the loop-through is enabled. Ideally, the VGA and 8514/A's LUTs should be set to the same values at all times the 8514/A has the loop-through enabled. The designers of the 8514/A took this into consideration and came up with a method in which both LUTs could be programmed simultaneously. Thus, any write to the VGA palettes while the 8514/A is displaying VGA information will result in an identical write to the 8514/A's LUT. Unlike the VGA's LUT, the 8514/A's can be read, which is handy if you need to save the state of the palette. It is also possible to write to the 8514/A's LUT without affecting the VGA's LUT.

ROM

The IBM 8514/A also contains ROM, but its only purpose is to provide some minor initialization information during the POST procedure required by the PS/2. It is doubtful that this will be necessary in ISA (Industry Standard Architecture — the bus that ATs use) versions of the 8514/A.

Registers

As we touched upon earlier, the core 8514/A technology is accessed by registers. Some of these registers, such as the Destination Y register, have multiple purposes, depending on which graphics command is being set up. One of the registers, the Multifunction Control register, is indexed, which means that a specific sequence of bits in the data be-

ing written contains an index that is used internally by the 8514/A to execute commands or set certain internal conditions.

As you may notice by looking at Table 6-1, IBM chose to use I/O addresses for the registers that generally have the lowest 10 bits in common. The reason for this is that most PC hardware did not decode I/O addresses beyond 10 bits, even though a full 16 bits of I/O address space were available. Apparently the feeling was that no one would need more than 1,024 different I/O addresses. Over time this has caused a lot of address-space cramping and I/O conflict. Therefore, IBM and other companies have found a workable solution by using only one or two addresses in the 10-bit address space, so the actual registers referenced can be differentiated by their more significant bits instead of by the least significant bits.

The CRT-control registers are used to determine the resolution and frequency with which graphical information will be presented and are relatively straightforward, as far as programming goes. Resolutions of 640 x 480 and 1,024 x 768 are supported by reprogramming these registers. At 1,024 x 768, the IBM 8514/A only runs interlaced, which induces a lot of flicker on the screen. The 640 x 480 mode runs flicker free.

The drawing-control registers support several different functions, but there are only a few critical registers that must be manipulated, after normal initialization, for every operation.

The Graphics Processor Status register shows the current state of the command queue. Up to eight commands can be queued. The types of commands that fill the queue include setting the x and y start positions, foreground and background raster operations (mixes), line-drawing parameters, and actual drawing requests. Applications should always check the state of the queue to ensure that enough entries are available for the drawing command about to be requested.

The Command register is used to request a graphics operation. The operations that can be requested include line drawing (once certain calculations have been made), rectangle fills, polygon fills (fast but tedious to set up because of their implementation), and short-stroke-vector drawing. This register also has flags to perform reads instead of writes, use PC data instead of color-register information as the source, do byte swapping, and perform 16-bit vs. 8-bit I/O.

Additionally, all data, such as that for images, is transferred through the Pixel Data Transfer register.

Another interesting feature of the 8514/A drawing processor is the wide range of mixes (raster operations/combinatorial functions). The most commonly used one is Source XOR Destination, which has long been used in computer graphics for cursors and rubber-banding lines because of the nature of an XOR to restore the original contents if applied twice with the same values. In lieu of traditional transparency, in which color 0 is considered transparent when used in an image or drawing, the 8514/A has something called a color comparison, in which a color is selected and the drawing operation is either performed or not, depending on the value of the destination pixel. Eight different comparisons are available, and if the result of the operation is TRUE, the pixel in memory is not updated. This approach allows for some interesting effects and graphical operations, because you can literally "write protect" a range of colors from being updated.

Register Specifications

Detailed descriptions of the IBM 8514/A registers follow, listed in ascending order according to their port addresses. All registers are assumed to be 16 bit unless otherwise noted.

Following the register address and register name is a representation of the entire 16-bit word, with a breakdown of its bit fields and an explanation of how each bit field affects the 8514/A's operation. Some

bit fields are more than one bit long, and we have provided tables with the binary values of those bits to show the breakdown of the field.

Table 6-1 lists the page numbers for each of the registers for those of you who need to look up a specific register. Table 6-2 lists the registers by category.

Table 6-1: The VESA-approved 8514/A register names, mnemonics, and addresses listed by register address

Port Address (hexadecimal)	Access	Mnemonic	Description
02E8	R	DISP_STAT	Display Status
02E8	W	H_TOTAL	Horizontal Total
02EA	R/W	DAC_MASK	DAC Mask
02EB	W	DAC_R_INDEX	DAC Read Index
02EC	W	DAC_W_INDEX	DAC Write Index
02ED	R/W	DAC_DATA	DAC Data
06E8	W	H_DISP	Horizontal Displayed
0AE8	W	H_SYNC_STRT	Horizontal Sync Start
0EE8	W	H_SYNC_WID	Horizontal Sync Width
12E8	W	V_TOTAL	Vertical Total
16E8	W	V_DISP	Vertical Displayed
1AE8	W	V_SYNC_STRT	Vertical Sync Start
1EE8	W	V_SYNC_WID	Vertical Sync Width
22E8	W	DISP_CNTL	Display Control
42E8	R	SUBSYS_STAT	Subsystem Status
42E8	W	SUBSYS_CNTL	Subsystem Control
46E8	W	ROM_PAGE_SEL	ROM Page Select
4AE8	W	ADVFUNC_CNTL	Advanced Function Control
82E8	R/W	CUR_Y	Current Y Position
86E8	R/W	CUR_X	Current X Position
8AE8	W	DESTY_AXSTP	Destination Y Position/Axial Step Constant
8EE8	W	DESTX_DIASTP	Destination X Position/Diagonal Step Constant
92E8	R/W	ERR_TERM	Error Term
96E8	W	MAJ_AXIS_PCNT	Major Axis Pixel Count

Port Address (hexa-decimal)	Access	Mnemonic	Description
9AE8	R	GP_STAT	Graphics Processor Status
9AF8	W	CMD	Command
9EE8	W	SHORT_STROKE	Short Stroke Vector Transfer
A2E8	W	BKGD_COLOR	Background Color
A6E8	W	FRGD_COLOR	Foreground Color
AAE8	W	WRT_MASK	Write Mask
AEE8	W	RD_MASK	Read Mask
B2E8	W	COLOR_CMP	Color Compare
B6E8	W	BKGD_MIX	Background Mix
BAE8	W	FRGD_MIX	Foreground Mix
BEE8		MULTIFUNC_CNTL	Multifunction Control
Index 0	W	MIN_AXIS_PCNT	Minor Axis Pixel Count
Index 1	W	SCISSORS_T	Top Scissors
Index 2	W	SCISSORS_L	Left Scissors
Index 3	W	SCISSORS_B	Bottom Scissors
Index 4	W	SCISSORS_R	Right Scissors
Index 5	W	MEM_CNTL	Memory Control
Index 8	W	PATTERN_L	Fixed Pattern Low
Index 9	W	PATTERN_H	Fixed Pattern High
Index A	W	PIX_CNTL	Pixel Control
E2E8	R/W	PIX_TRANS	Pixel Data Transfer

Table 6-2: The VESA-approved 8514/A register names, mnemonics, and addresses listed by category

Lookup-Table Registers

Port Address (Hexa-decimal)	Access	Mnemonic	Description
02EA	R/W	DAC_MASK	DAC Mask
02EB	W	DAC_R_INDEX	DAC Read Index
02EC	W	DAC_W_INDEX	DAC Write Index
02ED	R/W	DAC_DATA	DAC Data

Please note that all reserved bit fields should be explicitly set to 0 so as to avoid incompatibility with future versions of the 8514/A.

For examples of what 8514/A-register-based applications look like, refer to Listings 7-4 and after.

CRT-Control Registers

Port Address (Hexadecimal)	Access	Mnemonic	Description
02E8	R	DISP_STAT	Display Status
02E8	W	H_TOTAL	Horizontal Total
06E8	W	H_DISP	Horizontal Displayed
0AE8	W	H_SYNC_STRT	Horizontal Sync Start
0EE8	W	H_SYNC_WID	Horizontal Sync Width
12E8	W	V_TOTAL	Vertical Total
16E8	W	V_DISP	Vertical Displayed
1AE8	W	V_SYNC_STRT	Vertical Sync Start
1EE8	W	V_SYNC_WID	Vertical Sync Width
22E8	W	DISP_CNTL	Display Control
4AE8	W	ADVFUNC_CNTL	Advanced Function Control

Miscellaneous Control Registers

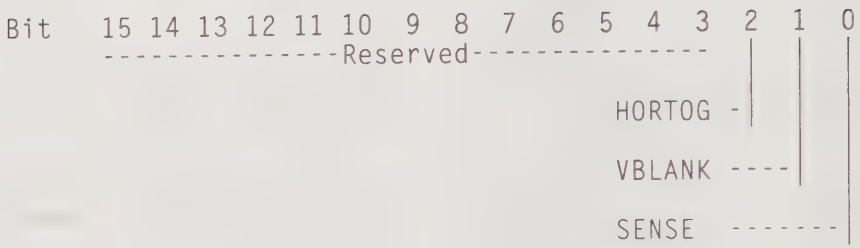
Port Address (Hexadecimal)	Access	Mnemonic	Description
42E8	R	SUBSYS_STAT	Subsystem Status
42E8	W	SUBSYS_CNTL	Subsystem Control
46E8	W	ROM_PAGE_SEL	ROM Page Select

Drawing-Control Registers

Port Address (Hexadecimal)	Access	Mnemonic	Description
82E8	R/W	CUR_Y	Current Y Position
86E8	R/W	CUR_X	Current X Position
8AE8	W	DESTY_AXSTP	Destination Y Position/Axial Step Constant
8EE8	W	DESTX_DIASTP	Destination X Position/Diagonal Step Constant

Port Address (Hexadecimal)	Access	Mnemonic	Description
92E8	R/W	ERR_TERM	Error Term
96E8	W	MAJ_AXIS_PCNT	Major Axis Pixel Count
9AE8	R	GP_STAT	Graphics Processor Status
9AE8	W	CMD	Command
9EE8	W	SHORT_STROKE	Short Stroke Vector Transfer
A2E8	W	BKGD_COLOR	Background Color
A6E8	W	FRGD_COLOR	Foreground Color
AAE8	W	WRT_MASK	Write Mask
AEE8	W	RD_MASK	Read Mask
B2E8	W	COLOR_CMP	Color Compare
B6E8	W	BKGD_MIX	Background Mix
BAE8	W	FRGD_MIX	Foreground Mix
BEE8		MULTIFUNC_CNTL	Multifunction Control
Index 0	W	MIN_AXIS_PCNT	Minor Axis Pixel Count
Index 1	W	SCISSORS_T	Top Scissors
Index 2	W	SCISSORS_L	Left Scissors
Index 3	W	SCISSORS_B	Bottom Scissors
Index 4	W	SCISSORS_R	Right Scissors
Index 5	W	MEM_CNTL	Memory Control
Index 8	W	PATTERN_L	Fixed Pattern Low
Index 9	W	PATTERN_H	Fixed Pattern High
Index A	W	PIX_CNTL	Pixel Control
E2E8	R/W	PIX_TRANS	Pixel Data Transfer

02E8 — Display Status Register
Read Only



HORTOG:
This bit changes state every time a horizontal sync is encountered. This means the state changes every line and repeats every other line.

VBLANK:
This bit is set during vertical blank.

SENSE:
This bit is set when the video data is sensed to be near peak (red, green, and blue values are at or near maximum value). This bit is also set when no monitor is attached.

Comments:
HORTOG and SENSE are used mainly for diagnostic purposes, whereas VBLANK is generally used to synchronize updates of the screen for animation purposes.

02E8 — Horizontal Total Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       -------Reserved--------  -- Horizontal Total --
```

The horizontal total is defined as the sum of the horizontal displayed data time, the horizontal sync time, and the horizontal front and back porches, less one. The unit of all horizontal timings is a double nugget. For example, in the IBM 1,024 x 768 mode, this register has a value of 158.

Comments:
This register is used to define the display frequency at which data is displayed on the attached monitor.

02EA — DAC Mask Register
Read/Write — 8 Bits

```
Bit      7  6  5  4  3  2  1  0
       --------- Mask ---------
```

The DAC Mask register is used to enable or disable individual bits of pixel data as they go into the LUT. For example, if you wanted to display only bits 0 and 1 of all the pixels on the screen, you would write 03H to this register.

Comments:
For the 16-color configuration, the default for this register is 0FH, and for the 256-color mode, FFH. This register is also readable. Additionally, when the VGA pass-through mode is enabled in the Advanced Function Control register, this register can be written at I/O address 03C6H.

02EB — DAC Read Index Register
Read/Write — 8 Bits

```
Bit     7  6  5  4  3  2  1  0
      ------ Read Index ------
```

The DAC Read Index register sets the position in the DAC from which the next LUT value is to be read via the DAC Data register.

Comments:
The main function of this register is to allow software to read the current contents of the DAC in order to be able to restore the DAC to its original state after the software has completed its task. This register is also readable. Additionally, when the VGA pass-through mode is enabled in the Advanced Function Control register, this register can be written at I/O address 03C7H.

02EC — DAC Write Index Register
Read/Write — 8 Bits

```
Bit      7  6  5  4  3  2  1  0
       ----- Write Index ------
```

The DAC Write Index register sets the position in the DAC to which the next LUT value is to be written via the DAC Data register.

Comments:
The main function of this register is to allow software to set the contents of the DAC. This register is also readable. Additionally, when the VGA pass-through mode is enabled in the Advanced Function Control register, this register can be written at I/O address 03C8H.

02ED — DAC Data Register
Read/Write — 8 Bits

```
Bit      7  6  5  4  3  2  1  0
       -------- Data ---------
```

The DAC Data register is used to write data to or read data from the DAC.

Comments:
This register is auto-incrementing. For example, if you have just set the DAC Read Index register, the first read of the DAC Data register would give you the red-gun value for the index you specified. The second and third reads would result in the blue and green values, respectively. Yet another read would cause an increment in the read-index number and return the red-gun value of new index. The same applies for writes.

The DAC Data register on the IBM 8514/A uses only the least significant 6 bits. The upper 2 bits are undefined, although some clone-board manufacturers may use them to get 256 shades per gun. This register is also readable. Additionally, when the VGA pass-through mode is enabled in the Advanced Function Control register, this register can be written at I/O address 03C9H.

06E8 — Horizontal Displayed Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       ---------Reserved---------  - Horizontal Disp.-
```

The horizontal displayed value is defined as the number of double nuggets on a given line, less one. For example, in the IBM 1,024 x 768 mode, this register has a value of 127, because in that mode, a double nugget consists of 8 pixels and 8 x 128 = 1,024.

Comments:
This register is one of those used to define the display frequency at which data is displayed on the attached monitor.

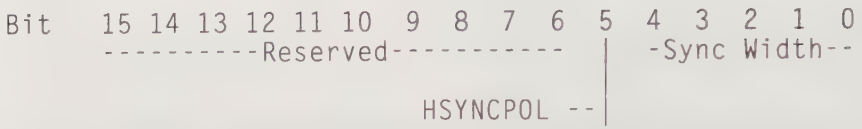
0AE8 — Horizontal Sync Start Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       -------Reserved--------  ----- Sync Start -----
```

The Horizontal Sync Start register specifies the start of the horizontal sync, relative to the start of displayed data, less one. This value is measured in double nuggets, and in the IBM 1,024 x 768 mode, this register has a value of 129.

Comments:
This register is one of those used to define the display frequency at which data is displayed on the attached monitor.

0EE8 — Horizontal Sync Width Register
Write Only



HSYNCPOL:
0 Horizontal sync has positive polarity
1 Horizontal sync has negative polarity

Sync Width:
The horizontal sync width specifies the width of the horizontal sync signal in double nuggets.

Comments:
The polarity setting is required by IBM monitors to determine which resolution to display. This register is one of those used to define the display frequency at which data is displayed on the attached monitor.

12E8 — Vertical Total Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       -Reserved-- ----- Vertical Total -----  -VTADJ-
```

Vertical Total:
This is the total number of rows per frame, less one. The definition of a row is dependent on several factors, including whether or not the display is interlaced.

VTADJ:
Vertical Total Adjust. This field allows for fine adjustment of the vertical total, specified in units of lines.

Comments:
This register is used to define the display frequency at which data is displayed on the attached monitor.

16E8 — Vertical Displayed Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       -Reserved-- ----- Vertical Disp. -----  -VDADJ-
```

Vertical Disp.:
This is the total number of displayed rows per frame, less one. The definition of a row is dependent on several factors, including whether or not the display is interlaced.

VDADJ:
Vertical Displayed Adjust. This field allows for fine adjustment of the vertical displayed count, specified in units of lines.

Comments:
This register is one of those used to define the display frequency at which data is displayed on the attached monitor.

1AE8 — Vertical Sync Start Register
Write Only

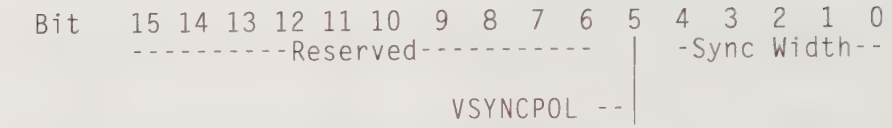
```
Bit   15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
      -Reserved-- ------- Sync Start -------  -VSADJ-
```

Sync Start:
This is the row position at which the vertical sync starts, relative to the beginning of the vertical displayed data. The definition of a row is dependent on several factors, including whether or not the display is interlaced.

VSADJ:
Vertical Sync Start Adjust. This field allows for fine adjustment of the vertical sync start, specified in units of lines.

Comments:
This register is one of those used to define the display frequency at which data is displayed on the attached monitor.

1EE8 — Vertical Sync Width Register
Write Only

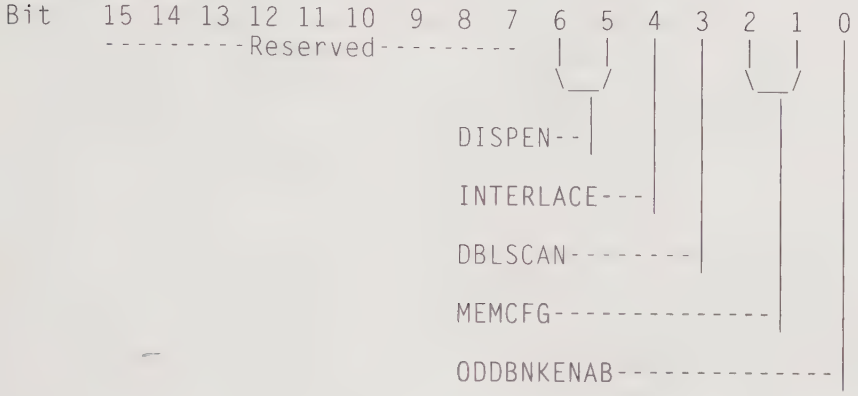


VSYNCPOL:
0 Vertical sync has positive polarity
1 Vertical sync has negative polarity

Sync Width:
The vertical sync width specifies the width of the vertical sync signal in rows. The definition of a row is dependent on several factors, including whether or not the display is interlaced.

Comments:
The polarity setting is required by IBM monitors to determine which resolution to display. This register is one of those used to define the display frequency at which data is displayed on the attached monitor.

22E8 — Display Control Register
Write Only



DISPEN:
This field controls display generation and video-memory refresh, as follows:

Bits 65	Description
00	No change from current setting
01	Enable display, syncs, and refresh
10, 11	Disable display, syncs, and refresh

INTERLACE:
0 Noninterlaced display
1 Interlaced display. Even lines appear in one field, and odd lines appear in another field. The fields are alternately displayed, allowing for the use of a slower and more inexpensive monitor.

DBLSCAN:
0 Display each scan line once
1 Display each scan line twice, thereby doubling the displayed vertical resolution

MEMCFG:
Memory Configuration. This field is used to define the row configuration of memory attached to the 8514/A. The value here determines the scalar used in calculating the vertical sync values and is based on the number of rows of memory. It should be set the same as the VRTCFG field in the Memory Control register. (CAS means Column Address Select, and applies to the addressing and accessing of RAM.)

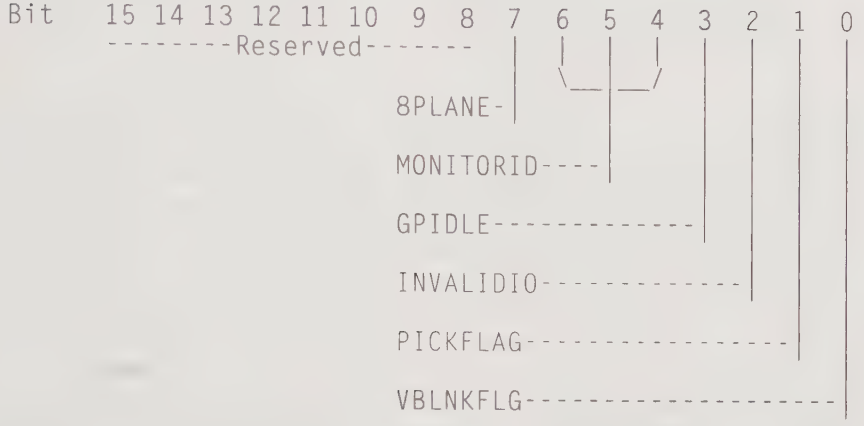
Bits 21	Description
00	2 CAS configuration
01	4 CAS configuration (default)
10	6 CAS configuration
11	8 CAS configuration

ODDBNKENAB:
Odd Bank Enable.

0	Use even banks only
1	Use alternate odd/even banks for each line (default)

Comments:
This register is vital to the proper initialization of the 8514/A. Some of its values also determine the scale of the vertical sync parameters.

42E8 — Subsystem Status Register
Read Only



8PLANE:
This bit indicates whether or not the 8514/A memory expansion has been installed, as follows:

0 4 planes, memory expansion installed
1 8 planes, memory expansion installed

MONITORID:
This field returns information from the three monitor-ID bits in IBM monitors. Each monitor type has a unique ID, as follows:

Bits 654	Description
000	Reserved
001	IBM 8507 (monochrome, 1,024 x 768 interlaced)
010	IBM 8514 (color, 1,024 x 768 interlaced)
011	Reserved
100	Reserved
101	IBM 8503 (monochrome, 640 x 480 noninterlaced)
110	IBM 8512 and IBM 8513 (color, 640 x 480 noninterlaced)
111	Reserved

GPIDLE:
This bit is set when the graphics processor is idle and the command queue is empty. This flag is cleared only by setting the RGPIDLE bit in the Subsystem Control register.

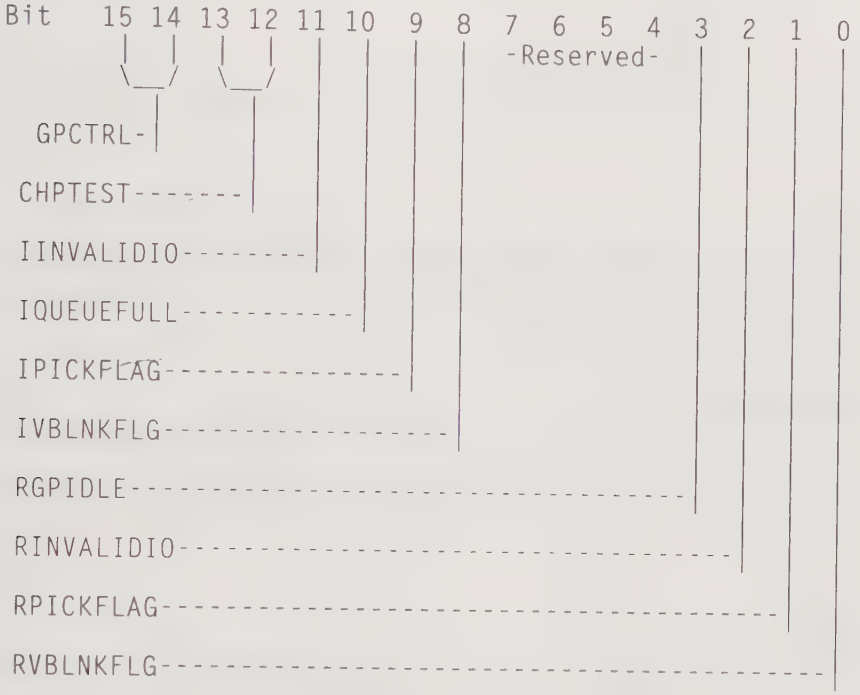
INVALIDIO:
This flag is set when (1) a command is written to a full queue or (2) the Pixel Data Transfer register was read without any data being available to read. This bit must be cleared by setting the RINVALIDIO bit in the Subsystem Control register prior to any other operation.

PICKFLAG:
This bit is set when a write to a pixel within the clipping rectangle is about to be made. This bit is cleared by setting the RPICKFLAG bit in the Subsystem Control register.

VBLNKFLG:
This bit is set at the start of the vertical-blanking period and is cleared only by setting the RVBLNKFLG bit in the Subsystem Control register.

Comments:
This register is used to gain information about the initial state of the 8514/A as well as information pertaining to the ongoing operation of the board.

42E8 — Subsystem Control Register
Write Only



GPCTRL:
This field is used to reset the 8514/A, as follows:

Bits 11 54	Description
00	No change
01	Enable 8514/A
10, 11	Reset and disable 8514/A; also flushes the command queue

CHPTEST:
This field is used to synchronize the two 8514/A chips and is used to test the chip communication. The field is defined as follows:

Bits 11 32	Description
00	No change
01	Enable synchronization between chips
10, 11	Disable synchronization; **should be done only as a diagnostic procedure**

IGPIDLE:
Enable GPIDLE Interrupt. If this bit is set, the 8514/A will interrupt the system when the GPIDLE bit in the Subsystem Status register goes high. IRQ9 (software IRQ2) is used to interrupt the system.

IINVALIDIO:
Enable INVALIDIO Interrupt. If this bit is set, the 8514/A will interrupt the system when the INVALIDIO bit in the Subsystem Status register goes high. IRQ9 (software IRQ2) is used to interrupt the system.

IPICKFLAG:
Enable PICKFLAG Interrupt. If this bit is set, the 8514/A will interrupt the system when the PICKFLAG bit in the Subsystem Status register goes high. IRQ9 (software IRQ2) is used to interrupt the system.

IVBLNKFLG:
Enable VBLNKFLG Interrupt. If this bit is set, the 8514/A will interrupt the system when the VBLNKFLG bit in the Subsystem Status register goes high. IRQ9 (software IRQ2) is used to interrupt the system.

RGPIDLE:
Setting this bit will reset the GPIDLE bit in the Subsystem Status register.

RINVALIDIO:
Setting this bit will reset the INVALIDIO bit in the Subsystem Status register.

RPICKFLAG:
Setting this bit will reset the PICKFLAG bit in the Subsystem Status register.

RVBLNKFLG:
Setting this bit will reset the VBLNKFLG bit in the Subsystem Status register.

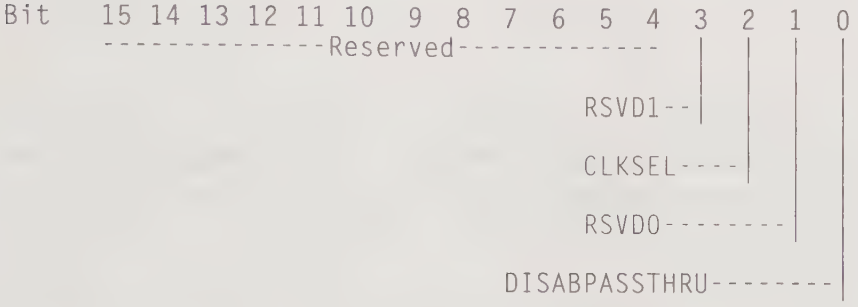
Comments:
This register is used to reset the 8514/A, control interrupts issued by the board, and clear the status flags.

46E8 — ROM Page Select Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       ---------------Reserved---------------  ROMPAGE
```

ROMPAGE:
This field is used to select which 4K page of the 32K ROM on-board the IBM 8514/A is mapped to memory. At present, the IBM 8514/A uses page 7. The page is mapped with the lower 2K at PC address CA000H and the upper 2K at PC address C6800H. Note that this applies only on a PS/2. Clone-board manufacturers may not use ROM at all. If ROM is used on PC-bus-compatible 8514/A clones, it must be mapped with care because of potential conflict with VGA ROMs.

4AE8 — Advanced Function Control Register
Write Only



RSVD1, RSVD0:
These bits are each connected to an undedicated output pin on the 8514/A chip set. Some of the clone boards use these pins to select additional clocks for display generation (see CLKSEL below). **Note: IBM always sets RSVD0 to 1, so this affects the setting this register.**

CLKSEL:
This bit selects which of the two clocks is to be used to generate the display signals, as follows:

0 Select the 25.175-MHz clock; used for a 640 x 480 display
1 Select the 44.9-MHz clock; used for a 1,024 x 768 display

DISABPASSTHRU:
0 Enable display of VGA pass-through information, and disable display of 8514/A information; also enable alternate addressing of the DAC register
1 Enable display of 8514/A information, and disable both the display of VGA information and the alternate DAC addressing

82E8 — Current Y Position Register
Read/Write

```
Bit   15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
      -Reserved-- ---------- Y Position ----------
```

This register can be read to get the current Y position or written to set the current Y position. The Y position is used by all the drawing functions to determine where to start drawing.

Comments:
Reading this register while a graphics operation is in progress may result in useless information because the value of this register will continue to change until the operation is complete.

86E8 — Current X Position Register
Read/Write

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       -Reserved-- ---------- X Position ------------
```

This register can be read to get the current X position or written to set the current X position. The X position is used by all the drawing functions to determine where to start drawing. The upper font bits of this register must be clear, or unpredictable drawing sequences may result.

Comments:
Reading this register while a graphics operation is in progress may result in useless information because the value of this register will continue to change until the operation is complete.

8AE8 - Destination Y Position/Axial Step Constant Register

Write Only

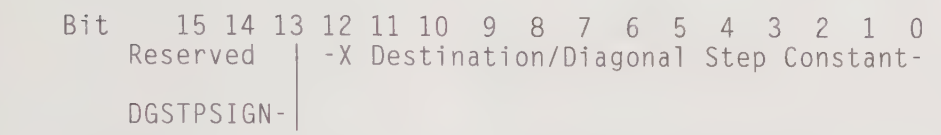
AXSTPSIGN:
This is the sign bit for the axial step constant. When the axial step constant is in use (see below), this bit, when set, indicates that the constant should be interpreted as a negative two's-complement number.

Y Destination/Axial Step Constant:
This field is used as the Y destination position when performing a BitBLT. For line-drawing operations, this field contains the axial step constant, a value required by the line-drawing routine used by the 8514/A. The axial step constant is calculated as:

2 x (minor axis delta)

Comments:
More information on calculating the value of this register is given in Chapter 7.

8EE8 —Destination X Position/Diagonal Step Constant Register
Write Only



DGSTPSIGN:
This is the sign bit for the diagonal step constant. When the diagonal step constant is in use (see below), this bit, when set, indicates that the constant should be interpreted as a negative two's-complement number.

X Destination/Diagonal Step Constant:
This field is used as the X destination position when performing a BitBLT. For line-drawing operations, this field contains the diagonal step constant, a value required by the line-drawing routine used by the 8514/A. The diagonal step constant is calculated as:

[2 x (minor axis delta)] – [2 x (major axis delta)]

Comments:
More information on calculating the value of this register is given in Chapter 7.

92E8 — Error Term Register
Read/Write

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       Reserved ----------Initial Error Term----------
```

Initial Error Term:
This register is used in line drawing on the 8514/A. The error term is calculated as:

[2 x (minor axis delta)] – (major axis delta) – fixup

Fixup is a value of either 0 or 1, depending on the direction of the line, and is used to allow for retracing lines even if coordinates are swapped. The contents of this register may be altered by the line-drawing command.

Comments:
More information on calculating the value of this register is given in Chapter 7. This register is also used to detect the presence of the IBM 8514/A by both the IBM AI and the ROM, because it can be read back.

96E8 — Major Axis Pixel Count Register
Write Only

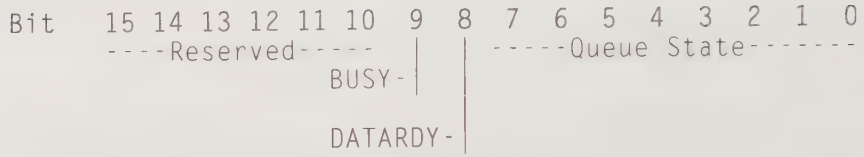
```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       ---Reserved--- -----Major Axis Pixel Count-----
```

Major Axis Pixel Count:
This register is used by the line-drawing, fill-rectangle, and BitBLT commands to specify the dimension of the major axis. The value for this register is calculated as:

major axis pixel count – 1

Comments:
More information on the use of this register is provided in the programming examples in Chapter 7.

9AE8 — Graphics Processor Status Register
Read Only



BUSY:

0 8514/A idle — no graphics command being executed

1 8514/A in the process of executing a graphics command. During data-transfer operations, this bit will remain set until all data has been transferred. Regular drawing operations also keep this bit set until they have been completed.

DATARDY:

0 No data waiting to be read from the Pixel Data Transfer register

1 Data waiting to be read from the Pixel Data Transfer register. This will happen only when a drawing operation is specified to read data from 8514/A display memory and give it to the PC (PCDATA is set and WRTDATA is cleared when the Command register is programmed with a drawing operation).

Queue State:

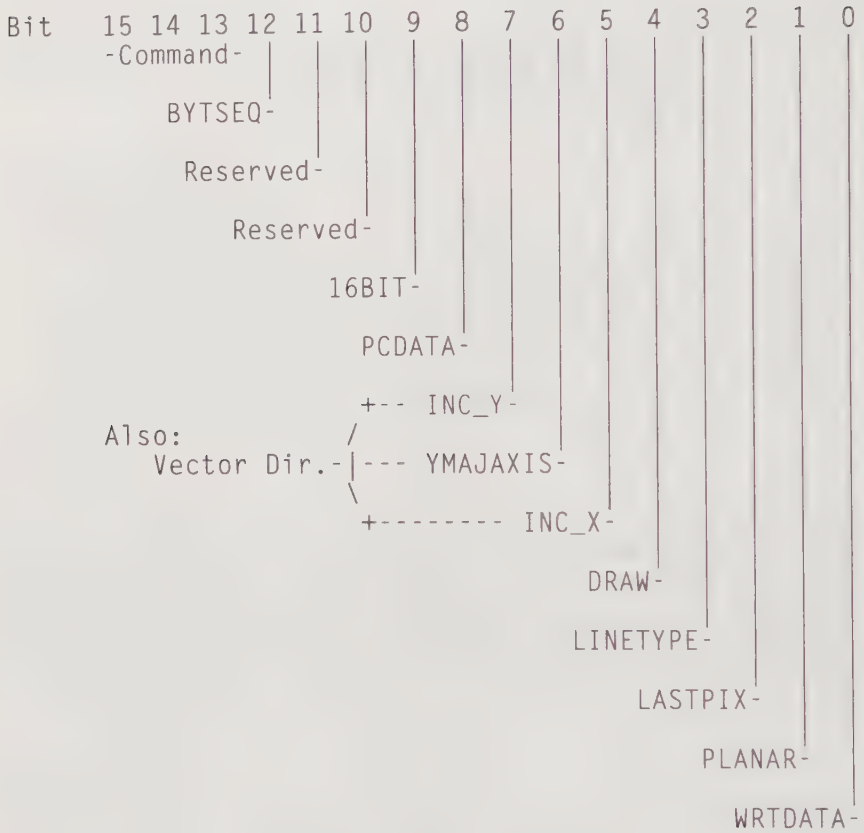
There are eight slots in the 8514/A command queue, as referenced by the 8 bits in the queue-state portion of this register. Setting virtually any register on the 8514/A is considered to be a queueable command, but in general, most operations, such as setting a color, are executed so quickly that unless a long drawing operation is in progress and therefore keeping the 8514/A busy, they will never be visible in the queue.

The interpretations of the queue-state bits are as follows:

Queue State Value (binary)	Meaning
00000000	Queue empty (This does not imply that the 8514/A is idle. A command may still be executing. This can be determined by checking the BUSY bit.)
00000001	7 entries available
00000011	6 entries available
00000111	5 entries available
00001111	4 entries available
00011111	3 entries available
00111111	2 entries available
01111111	1 entry available
11111111	Queue full

Comments:
This is one of the most important registers in the 8514/A. If the queue state is not checked, it is possible to write too much data to the 8514/A, resulting in a potential deadlock situation that can be cleared only by setting the RINVALIDIO bit in the Subsystem Control register.

9AE8 — Command Register
Write Only



Command:
This 3-bit field indicates the type of drawing operation being requested. All functions that actually draw use the Current X and Y Position registers as the starting points for the drawing operation.

Also, all operations are subject to the current color, mixes, color compares, plane masks, and clipping rectangle. The available types of operations are:

Bits 111 543	Description
000	No Operation Performed. This is used to enable short-stroke-vector mode without accidentally drawing anything in display memory.
001	Line Draw. This operation will draw a line in one of two ways, depending on the state of the LINETYPE bit. In addition to using the Major Axis Pixel Count register, line drawing uses other registers determined by the state of the LINETYPE bit. If the LINETYPE bit is cleared to 0, the line drawn will also be based on the values of the Axial Step Constant, Diagonal Step Constant, and Error Term registers and the INC_Y, YMAJAXIS, and INC_X bits. If the LINETYPE bit is set to 1, bits 5 – 7 of the Command register specify the direction (in 45-degree increments) in which the line is drawn. The length of such a line is determined by the contents of the Major Axis Pixel Count register. The translation of bits 5, 6, and 7 into directions is as follows:

Bits 765	Direction in Degrees (counterclockwise from positive X axis)
000	0
001	45
010	90
011	135
100	180
101	225
110	270
111	315

The current position is updated after this operation.

010 Fast-Fill Rectangle. This operation also has two possible uses, determined by the setting of the PLANEMODE bit in the Pixel/Control register. In either case, the size of the rectangle being accessed/drawn is determined by the contents of the Major Axis Pixel Count register, which indicates the X width of the rectangular region, and the Minor Axis Pixel Count register, which defines the number of lines, or Y height, of the region. If the PLANEMODE bit is cleared to 0, this operation draws a solid (filled) rectangle in display memory. If the PLANEMODE bit is set, the 8514/A uses the Read Mask register to determine which plane of memory is to be used as a source for an area fill. The area fill scans the selected plane for enabled bits. Every time it finds a set bit, it toggles an internal-fill flag. During the scanning, if the fill flag is set, a pixel is written through all write-enabled planes at the point that was just scanned. This is a quick way to fill objects, but it is limited in use because you must dedicate a plane to it, thereby potentially losing color content. This operation can also be used to load data into display memory by enabling the PCDATA bit in the Command register.

The current position is updated after this operation.

011 Fill Rectangle Vertically (#1). This operation is similar to the previous one but operates vertically, not horizontally. The Minor Pixel Count register specifies the number of vertical lines in the fill area, and the Major Axis Pixel Count register specifies the number of pixels in each vertical line. The filling/drawing commences at the top of a line and works its way down.

The current position is updated after this operation.

100 Fill Rectangle Vertically (#2). This is somewhat similar to the previous operation. The major difference is that pixels are referenced four at a time horizontally as the fill/draw progresses

vertically. At the end of each vertical strip, the horizontal position is updated and the fill/draw proceeds in the opposite direction. It can get tricky using PC data as the pixel source for these operations.

The current position is *not* updated after this operation.

101 Draw Line for Area Fill. This function is similar to the line-drawing function except that it draws only one pixel for each scan line the drawn line traverses. This is useful for setting up an area fill.

The current position is updated after this operation.

110 Copy Rectangle. This is the BitBLT (bit-block transfer) operation. It is used to copy data from one part of display memory to another. It can also be used to transfer monochrome data from the PC to the 8514/A and expand it into color. This operation utilizes the Destination X and Y, Major Axis Pixel Count, and Minor Axis Pixel Count registers.

The current position is *not* updated after this operation.

111 Reserved

BYTSEQ:
Byte Sequence affects the order of bytes when writing them a word (16 bits) at a time. This applies only to the Pixel Data Transfer and Short Stroke Vector Transfer registers but not any other 16-bit register. It is valid only when the 16BIT flag is set.

0 High byte first, low byte second
1 Low byte first, high byte second

16BIT:
16 bit or 8 bit determines whether or not 16 bits can be written at a time. This applies only to the Pixel Data Transfer and Short Stroke Vector Transfer registers. It does not affect 16-bit registers, as these must always be accessed 16 bits at a time.

0 8-bit access
1 16-bit access

PCDATA:
Use PC Data enables/disables transfer of PC data.

0 Drawing operations use 8514/A-based data
1 Drawing operations wait for data to be written to or read from the Pixel Data Transfer register before proceeding to the next pixel. The direction of the transfer depends on the state of the WRTDATA bit.

INC_Y:
Increment Y determines the Y direction of lines during line drawing when LINETYPE is cleared.

0 Lines drawn in negative Y direction (up)
1 Lines drawn in positive Y direction (down)

YMAJAXIS:
Y Major Axis determines which axis is the major axis for line drawing when LINETYPE is 0.

0 X is major axis
1 Y is major axis

INC_X:
Increment X determines the X direction of lines during line drawing when LINETYPE is 0.

0 Lines drawn in negative X direction (left)
1 Lines drawn in positive X direction (right)

DRAW:
Draw or Move chooses between drawing (modifying data in the display buffer) or moving the current position, based on the operation chosen.

0 Move only; no pixels drawn
1 Draw pixels

LINETYPE:
Line Type chooses between a Bresenham line draw (any direction) or directional vector.

0 Select normal Bresenham line drawing
1 Select vector drawing. If this bit is set and a line-drawing operation is selected at the same time, a vector will be drawn as described under the line-drawing operation. If no operation is selected, this bit enables short-stroke-vector drawing. All short-stroke-vector data is specified via the Short Stroke Vector Transfer register.

LASTPIX:
Last Pixel Drawn determines whether or not the last pixel of line draws and vector draws is drawn. This is used for polylines drawn in XOR mode and for setting up the 8514/A area fill.

0 Last pixel for lines and vectors drawn
1 Last pixel not drawn

PLANAR:
Planar Access specifies if data is accessed one pixel at a time or in a planar fashion (a nugget at a time). It is used primarily for monochrome pixels written from the PC to the 8514/A. In the planar mode, some graphics operations also speed up. Operations such as line drawing, rectangle fills, and BitBLTs benefit from the increased throughput by a factor of up to four times over using the pixel mode, because the planar mode allows for transfer of up to a nugget at a time.

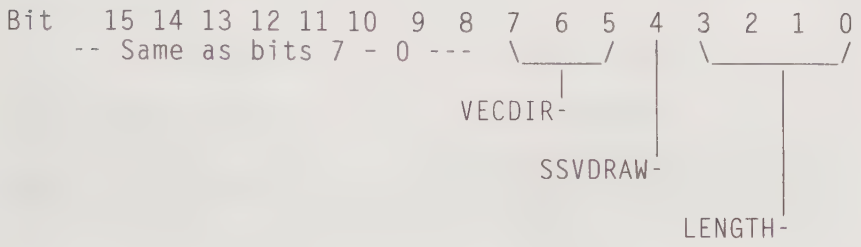
0 One pixel at a time
1 Planar/nugget mode

WRTDATA:
Write or Read Data specifies data will be read or written.

0 Read data from display memory; used to transfer board-based image data to system memory
1 Write data; normal way to do things

Comments:
This register is the sole register responsible for initiating all drawing operations on the 8514/A.

9EE8 — Short Stroke Vector Transfer Register
Write Only



VECDIR:
Vector Director is the direction of the short-stroke vector in degrees, as measured counterclockwise from the positive X axis. The values are defined as follows:

Bits	Direction in Degrees
765	(counterclockwise from positive X axis)
000	0
001	45
010	90
011	135
100	180
101	225
110	270
111	315

SSVDRAW:
Action determines whether only the current position is moved or whether pixels are drawn as well.

0 Move current position
1 Draw vector and move current position

LENGTH:
This is the length of the vector in pixels; it ranges from 0 to 15 in value.

Comments:
This register is affected by the state of the BYTSEQ flag in the Command register. Short strokes are also used to define the short-stroke fonts shipped with the IBM 8514/A. A routine that loads these fonts and uses them is presented in Chapter 7.

A2E8 — Background Color Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       -------Reserved--------  ---Background Color---
```

Background Color:
This register contains the background color used with certain values of the Foreground and Background Mix registers, as well as the Fixed Pattern registers (in the Multifunction Control register).

Comments:
Reading this register while no command is active may cause the 8514/A to freeze until the RINVALIDIO bit is set in the Subsystem Control register.

A6E8 — Foreground Color Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       -------Reserved--------  ---Foreground Color---
```

Foreground Color:
This register contains the foreground color used in normal drawing operations.

Comments:
Reading this register while no command is active may cause the 8514/A to freeze until the RINVALIDIO bit is set in the Subsystem Control register.

AAE8 — Write Mask Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       -------Reserved--------  ------Write Mask------
```

Write Mask:
This field controls which planes of the four or eight on the 8514/A board can be updated. For each bit in the field that is set, the corresponding plane is enabled for updating. If that bit is cleared, the corresponding plane is not updated during a pixel-write operation. In a four-plane configuration, only bits 3 – 0 are valid.

Comments:
The contents of this register will affect the operation of all mix commands.

AEE8 — Read Mask Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       -------Reserved--------  ------Read Mask-------
```

Read Mask:
This field controls which plane is used as source data for planar commands (those specified by the PLANEMODE bit in the Pixel Control register). This function is used in two different scenarios. The first is when either a draw command with WRTDATA cleared (read pixels) or a BitBLT command is used in conjunction with the PC being the destination. In such a situation, a set bit in the read mask enables a certain plane for reading, as follows:

Bit	Plane
0	7
1	0
2	1
3	2
4	3
5	4
6	5
7	6

The other use of the read mask is for the area-fill command (see the Command register for more information on how to set this up). In this case, the bits in the read mask correspond to the selected plane in the following manner:

Bit	Plane
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7

Note that the two uses of the read mask differ in the planes they reference, so they can be used at the same time without serious conflict.

Comments:
This register should be set to FFH at all times other than when it is specifically in use, because if the PLANEMODE bit in the Pixel Control register happens to be set during a normal BitBLT operation, only enabled planes will be used for source in mix calculations.

B2E8 — Color Compare Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       -------Reserved--------  ----Color Compare-----
```

Color Compare:
This field contains the color to be used in color comparisons as per the setting of the COLCMPOP field in the Pixel Control register. During a normal pixel-update operation, the color-compare operation is used to determine whether or not a given pixel should be updated, based on its color. The destination-pixel color is compared to the color-compare field using the color-compare operation, and if the result of the comparison is TRUE, the pixel is not updated. The current write mask determines which planes are used during the color-compare. Planes that are write-disabled are treated as 0s in the comparison.

B6E8 — Background Mix Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       ---------Reserved---------  BSS-  -----MIX-----
```

BSS:
Background Source Select. This field is used to select the source of data for background mix operations. The background mix is used when patterns are selected (see the description of the Fixed Pattern registers), as well as during some BitBLT operations (as specified in the Pixel Control register).

For background mix source data, use:

Bits

65	Description
00	Background color
01	Foreground color
10	PC data (via Pixel Data Transfer register)

11 All-Plane Copy. This selects all active planes of the source data for BitBLTing to the destination area. This use is meant to be selected only in conjunction with the BitBLT operation.

MIX:
This field specifies the background mix to use for those operations that require it. A mix combines the source pixel (as determined by the BSS field) with a destination pixel (the one in display memory) to produce a new destination pixel, as follows:

Value	Mix
00H	NOT Dst
01H	0 (all bits cleared)
02H	1 (all bits set)
03H	Dst
04H	NOT Src
05H	Src XOR Dst
06H	NOT (Src XOR Dst)
07H	Src
08H	NOT Src OR NOT Dst
09H	NOT Src OR Dst
0AH	Src OR NOT Dst
0BH	Src OR Dst
0CH	Src AND Dst
0DH	Src AND NOT Dst
0EH	NOT Src AND Dst
0FH	NOT Src AND NOT Dst
10H	MINIMUM(Src, Dst)
11H	Dst – Src [with underflow]
12H	Src – Dst [with underflow]
13H	Src + Dst [with overflow]
14H	MAXIMUM(Src, Dst)
15H	(Dst – Src)/2 [with underflow]
16H	(Src – Dst)/2 [with underflow]
17H	(Src + Dst)/2 [with overflow]
18H	Dst – Src [with saturate]
19H	Dst – Src [with saturate, same as #18H]
1AH	Src – Dst [with saturate]
1BH	Src + Dst [with saturate]
1CH	(Dst – Src)/2 [with saturate]
1DH	(Dst – Src)/2 [with saturate, same as #1CH]
1EH	(Src – Dst)/2 [with saturate]
1FH	(Src + Dst)/2 [with saturate]

Src means the source pixel from the color registers, PC, or 8514/A memory. *Dst* means the destination pixel that already resides in 8514/A memory. AND, OR, XOR, and NOT are all bitwise binary operators. *Saturate,* as used above, means that the operations are bounded at 0 in the case of subtractions or at the maximum pixel value in the case of additions.

Comments:
The Write Mask and Color Compare registers supersede mixes.

BAE8 — Foreground Mix Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       ---------Reserved---------  FSS-  -----MIX-----
```

FSS:
Foreground Source Select. This field is used to select the source of data for foreground mix operations. The foreground mix is used at all times, unless the background mix is active for a given pixel.

For foreground mix source data, use:

Bits 65	Description
00	Background color
01	Foreground color
10	PC data (via Pixel Data Transfer register)
11	All-Plane Copy. This selects all active planes of the source data for BitBLTing to the destination area. This use is meant to be selected only in conjunction with the BitBLT operation.

MIX:

This field specifies the foreground mix to use for those operations that require it. A mix combines the source pixel (as determined by the FSS field) with a destination pixel (the one in display memory) to produce a new destination pixel, as follows:

Value	Mix
00H	NOT Dst
01H	0 (all bits cleared)
02H	1 (all bits set)
03H	Dst
04H	NOT Src
05H	Src XOR Dst
06H	NOT (Src XOR Dst)
07H	Src
08H	NOT Src OR NOT Dst
09H	NOT Src OR Dst
0AH	Src OR NOT Dst
0BH	Src OR Dst
0CH	Src AND Dst
0DH	Src AND NOT Dst
0EH	NOT Src AND Dst
0FH	NOT Src AND NOT Dst
10H	MINIMUM(Src, Dst)
11H	Dst – Src [with underflow]
12H	Src – Dst [with underflow]
13H	Src + Dst [with overflow]
14H	MAXIMUM(Src, Dst)
15H	(Dst – Src)/2 [with underflow]
16H	(Src – Dst)/2 [with underflow]
17H	(Src + Dst)/2 [with overflow]
18H	Dst – Src [with saturate]
19H	Dst – Src [with saturate, same as #18H]
1AH	Src – Dst [with saturate]
1BH	Src + Dst [with saturate]
1CH	(Dst – Src)/2 [with saturate]
1DH	(Dst – Src)/2 [with saturate, same as #1CH]
1EH	(Src – Dst)/2 [with saturate]
1FH	(Src + Dst)/2 [with saturate]

Src means the source pixel from the color registers, PC, or 8514/A memory. *Dst* means the destination pixel that already resides in 8514/A memory. AND, OR, XOR, and NOT are all bitwise binary operators. *Saturate*, as used above, means that the operations are bounded at 0 in the case of subtractions or at the maximum pixel value in the case of additions.

Comments:
The Write Mask and Color Compare registers supersede mixes.

BEE8 — Multifunction Control Register
Write Only

```
Bit   15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
      ---Index--- ------Register-Specific Data-------
```

This is one of the more peculiar 8514/A registers. The most significant 4 bits actually specify another register, with the remaining 12 bits specifying the data for that register. To simplify the explanation of these "subregisters," we have listed each one of them separately. A summary of the indices and what they refer to is as follows:

Index	Register
0	Minor Axis Pixel Count
1	Top Scissors
2	Left Scissors
3	Bottom Scissors
4	Right Scissors
5	Memory Control
8	Fixed Pattern Low
9	Fixed Pattern High
A	Pixel Control
6, 7, B, C, D, E, F	Reserved

BEE8 Index 0 — Minor Axis Pixel Count Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
        0  0  0  0 ------Minor Axis Pixel Count-------
```

Minor Axis Pixel Count:
This register is used by the fill-rectangle and BitBLT commands to specify the dimension of the minor axis. The value for this register is calculated as:

minor axis pixel count – 1

Comments:
More information on the use of this register is provided in the programming examples in Chapter 7.

BEE8 Index 1 — Top Scissors Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
        0  0  0  1 ----------Top Scissors-----------
```

Top Scissors:
This field is used to set the top edge of the clipping rectangle. The top is the lowest, numerically, vertical position (that is, minimum Y) at which pixels can be updated. To draw in all drawable memory, this field should be set to 0.

Comments:
More information on the use of this register is provided in the programming examples in Chapter 7.

BEE8 Index 2 — Left Scissors Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
        0  0  1  0 ----------Left Scissors----------
```

Left Scissors:
This field is used to set the left edge of the clipping rectangle. The left is the lowest, numerically, horizontal position (that is, minimum X) at which pixels can be updated. To draw in all drawable memory, this field should be set to 0.

Comments:
More information on the use of this register is provided in the programming examples in Chapter 7.

BEE8 Index 3 — Bottom Scissors Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
        0  0  1  1 ---------Bottom Scissors-----------
```

Bottom Scissors:
This field is used to set the bottom edge of the clipping rectangle. The top is the highest, numerically, vertical position (that is, maximum Y) at which pixels can be updated. To draw in all drawable memory, this field should be set to 3FFH.

Comments:
More information on the use of this register is provided in the programming examples in Chapter 7.

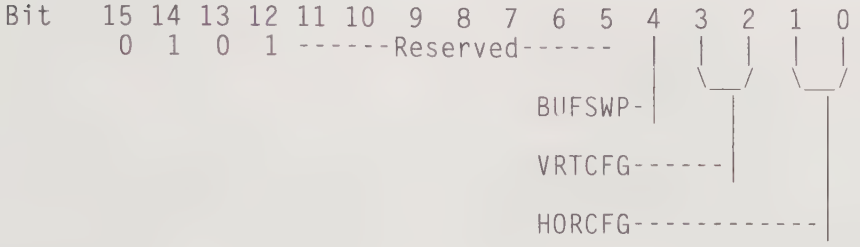
BEE8 Index 4 — Right Scissors Register
Write Only

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
        0  1  0  0 ----------Right Scissors----------
```

Right Scissors:
This field is used to set the right edge of the clipping rectangle. The right is the highest, numerically, horizontal position (that is, maximum X) at which pixels can be updated. To draw in all drawable memory, this field should be set to 3FFH.

Comments:
More information on the use of this register is provided in the programming examples in Chapter 7.

BEE8 Index 5 — Memory Control Register
Write Only



BUFSWP:
This bit is used during the pseudo 8-plane mode that can be configured on a 4-bit board at 640 x 480 resolution.

0 All reads and writes from/to buffer 0 (lower four planes)
1 All reads/writes from/to buffer 1 (upper four planes)

VRTCFG:
This field is used to tell the 8514/A chip set about the memory configuration it is attached to. The value set here affects the vertical sync parameters and should also be reflected in the MEMCFG field in the Display Control register. (CAS stands for Column Address Select.)

Bits

32	Description
00	2 CAS configuration
01	4 CAS configuration (default)
10	6 CAS configuration
11	8 CAS configuration

HORCFG:
This field also provides the 8514/A chip set with information about the memory architecture it is attached to, by setting the number of "write enables" it should use. This relates directly to the fact that horizontal sync parameters are measured in double nuggets. The set-

ting here affects the size of a double nugget. (RAS stands for Row Address Select.)

Bits 10	Description
00	4 write enables
01	5 write enables
10	8 write enables (default)
11	10 write enables (could be used for 1,280 x 1,024 resolution)

Comments:
This register is dependent on the existing hardware configuration and may need to be programmed differently on some clone boards.

BEE8 Index 8 — Fixed Pattern Low Register
Write Only

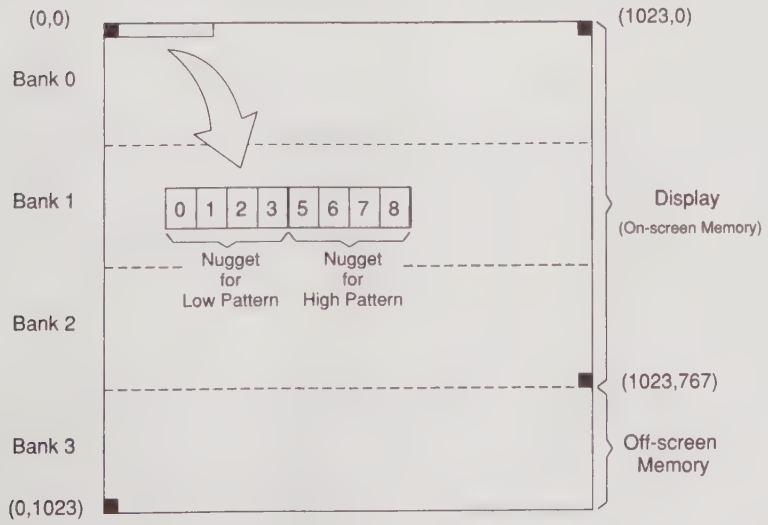
```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
        1  0  0  0 ------Reserved------  ---Mask Low--
```

Mask Low:
This field determines whether the foreground or background mix is used for a given pixel within the left half of a double nugget. Use of the pattern is enabled in the Pixel Control register. A set bit in the pattern selects the foreground mix, and a cleared bit selects the background mix for the specified pixel within a nugget during update operations. The mapping of the pattern mask to the pixels within the double nugget is as follows:

Bit	Pixel
0	4 (used only in 5-pixel nugget configurations)
1	3
2	2
3	1
4	0

Pixel 0 is the leftmost pixel in a double nugget.

Figure 6-5: 8514/A Video Timing Registers



BEE8 Index 9 — Fixed Pattern High Register
Write Only

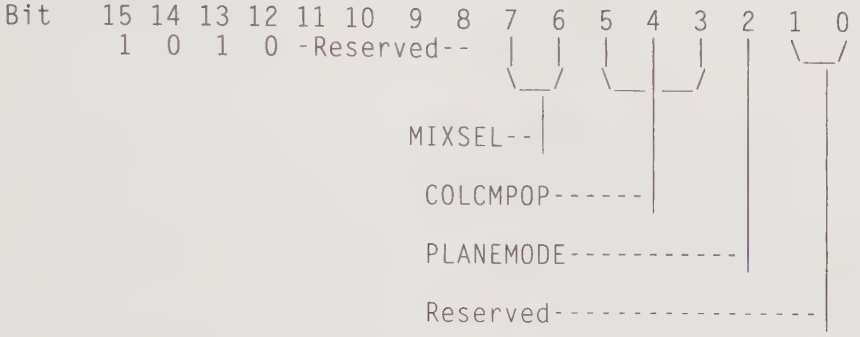
```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
        1  0  0  1 ------Reserved------  --Mask High--
```

Mask High:
This field determines whether the foreground or background mix is used for a given pixel within the right half of a double nugget. Use of the pattern is enabled in the Pixel Control register. A set bit in the pattern selects the foreground mix, and a cleared bit selects the background mix for the specified pixel within a nugget during update operations. The mapping of the pattern mask to the pixels within the double nugget is as follows:

Bit	Pixel
0	9 (used only in 5-pixel nugget configurations)
1	8
2	7
3	6
4	4

Pixel 8 (9 in a 5-pixel nugget configuration) is the rightmost pixel in a double nugget.

BEE8 Index A — Pixel Control Register
Write Only



MIXSEL:
Mix Selection. This field selects how mixes are to be used.

Bits
76 Description

00 Foreground Mix Active. Use the foreground mix for all drawing operations (default).

01 Fixed Pattern. Use the fixed pattern (Fixed Pattern registers) to decide which mix (foreground or background) is active for a given pixel within a double nugget.

10 PC Data Expansion. Use PC data in a fashion similar to the fixed pattern. See the description of the Pixel Data Transfer register for more information.

11 Single-Plane Expansion. This operates similar to the fixed pattern as well. The source plane is selected by the Read Mask register. The bits in the source plane specify whether the background (source bit = 0) or foreground mix (source bit = 1) is to be used to generate the destination pixel.

COLCMPOP:
Color-Compare Operation. This field is used to select the color-compare operation to be used on all pixels. See the description of the Color Compare register for more information.

Bits

543	Description
000	FALSE
001	TRUE
010	Dst >= CC
011	Dst < CC
100	Dst != CC
101	Dst == CC
110	Dst <= CC
111	Dst > CC

Dst refers to the destination pixel, and *CC* refers to the color specified in the Color Compare register. If the result of the operation is FALSE, the destination pixel is updated by the source pixel (or result of the mix, if one is in effect).

PLANEMODE:
When this bit is set, plane mode is enabled for area-fill and single-plane expansions (from any source). The source plane is selected via the Read Mask register. The default for this bit is to have it cleared.

Comments:
This register should be used with caution because it is possible to block all drawing commands with a bad setting.

E2E8 — Pixel Data Transfer Register
Read/Write

```
Bit    15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
       ----------------Pixel/Plane Data---------------
```

This register is the only way to transfer information to and from the IBM 8514/A. Transfers must be initiated by using a drawing command in the Command register with the PCDATA bit set or by specifying a PC data mix source. Two types of data can be transferred via this register: pixel and planar. The issues and differences are discussed next.

Pixel Transfers:
Pixel data is specified as 8 bits of color. If the 16BIT bit in the Command register is set, two pixels can be transferred at a time. The BYTSWP bit also affects the sequence of pixels in the 16BIT mode.

Planar Mode:
This mode can get somewhat complicated because the transferred data must be massaged to put it into a usable format. The data format on the 8514/A side is nugget-oriented, just like the Fixed Pattern registers are. This means that, ideally, the Current X Position register should be set to the start of a nugget when the transfer operation starts; otherwise, the PC software must rearrange data to conform to a midnugget transfer. The bits in the transfer word that correspond to pixels in the 8514/A nugget are as follows:

Bit	Pixel
0	4 (used only in 5-pixel nugget configurations)
1	3
2	2
3	1
4	0

Pixel 0 is the leftmost one in a nugget.

If the 16BIT bit is set in the Command register when the transfer operation is initiated, the same configuration in the chart above is reflected in bits 12 – 8 as well.

The planar mode is selected via the Pixel Control register.

Comments:
This register should be accessed only after a transfer request has been issued via the Command register. If a read access is inadvertently made from this register when no pertinent command is active in the 8514/A, then the 8514/A may freeze up. Such a freeze requires that the RINVALIDIO bit be set in the Subsystem Control register.

7

Tutorial: 8514/A Registers

With the reference information from the previous chapter, this chapter will help you start programming the 8514/A hardware. To ease the learning process, we shall parallel many of the programs from the AI tutorial in Chapter 5 so that you can see both the similarities and the differences between using the AI and the registers.

The examples provided here are written for use with Microsoft C Version 5.1 and MASM Version 5.1. Listing 7-1 contains a list of definitions we use in our C programs for register programming.

Listing 7-1: Important definitions used in all the register-based listings

```
/*
        REG8514.H

    Written by Jake Richter
    Copyright (c) 1989,1990 Panacea Inc., Londonderry, NH - All Rights Reserved

    This code may be freely incorporated in any program without
    royalty, as long as the copyright notice stays intact.

        This include file contains the standard definitions necessary for
    programming the IBM 8514/A.

        The following definitions are used in all the listings to reference
    certain data types. This is done to help make the code more readable,
    and to assist later translation and communication with assembly language.
*/

#define  FALSE          0
#define  TRUE           !(FALSE)

#define  BYTE           char
```

```
#define  WORD            short
#define  DWORD           long
#define  UBYTE           unsigned BYTE
#define  UWORD           unsigned WORD
#define  UDWORD          unsigned DWORD
#define  FAR             far
#define  HUGE            huge
#define  VOID            void

/*
    VESA Approved Register Definitions
*/
#define  DAC_MASK        0x02EA
#define  DAC_R_INDEX     0x02EB
#define  DAC_W_INDEX     0x02EC
#define  DAC_DATA        0x02ED
#define  DISP_STAT       0x02E8
#define  H_TOTAL         0x02E8
#define  H_DISP          0x06E8
#define  H_SYNC_STRT     0x0AE8
#define  H_SYNC_WID      0x0EE8
#define  V_TOTAL         0x12E8
#define  V_DISP          0x16E8
#define  V_SYNC_STRT     0x1AE8
#define  V_SYNC_WID      0x1EE8
#define  DISP_CNTL       0x22E8
#define  ADVFUNC_CNTL    0x4AE8
#define  SUBSYS_STAT     0x42E8
#define  SUBSYS_CNTL     0x42E8
#define  ROM_PAGE_SEL    0x46E8
#define  CUR_Y           0x82E8
#define  CUR_X           0x86E8
#define  DESTY_AXSTP     0x8AE8
#define  DESTX_DIASTP    0x8EE8
#define  ERR_TERM        0x92E8
#define  MAJ_AXIS_PCNT   0x96E8
#define  GP_STAT         0x9AE8
#define  CMD             0x9AE8
#define  SHORT_STROKE    0x9EE8
#define  BKGD_COLOR      0xA2E8
#define  FRGD_COLOR      0xA6E8
#define  WRT_MASK        0xAAE8
#define  RD_MASK         0xAEE8
#define  COLOR_CMP       0xB2E8
#define  BKGD_MIX        0xB6E8
#define  FRGD_MIX        0xBAE8
#define  MULTIFUNC_CNTL  0xBEE8
#define  MIN_AXIS_PCNT   0x0000
#define  SCISSORS_T      0x1000
#define  SCISSORS_L      0x2000
#define  SCISSORS_B      0x3000
```

```
#define   SCISSORS_R      0x4000
#define   MEM_CNTL        0x5000
#define   PATTERN_L       0x8000
#define   PATTERN_H       0x9000
#define   PIX_CNTL        0xA000
#define   PIX_TRANS       0xE2E8

/* Display Status Bit Fields */
#define   HORTOG          0x0004
#define   VBLANK          0x0002
#define   SENSE           0x0001

/* Horizontal Sync Width Bit Field */
#define   HSYNCPOL_NEG    0x0020
#define   HSYNCPOL_POS    0x0000

/* Vertical Sync Width Bit Field */
#define   VSYNCPOL_NEG    0x0020
#define   VSYNCPOL_POS    0x0000

/* Display Control Bit Field */
#define   DISPEN_NC       0x0000
#define   DISPEN_DISAB    0x0040
#define   DISPEN_ENAB     0x0020
#define   INTERLACE       0x0010
#define   DBLSCAN         0x0008
#define   MEMCFG_2        0x0000
#define   MEMCFG_4        0x0002
#define   MEMCFG_6        0x0004
#define   MEMCFG_8        0x0006
#define   ODDBNKENAB      0x0001

/* Subsystem Status Register */
#define   _8PLANE         0x0080
#define   MONITORID_8503  0x0050
#define   MONITORID_8507  0x0010
#define   MONITORID_8512  0x0060
#define   MONITORID_8513  0x0060
#define   MONITORID_8514  0x0020
#define   MONITORID_NONE  0x0070
#define   MONITORID_MASK  0x0070
#define   GPIDLE          0x0008
#define   INVALIDIO       0x0004
#define   PICKFLAG        0x0002
#define   VBLNKFLG        0x0001

/* Subsystem Control Register */
#define   GPCTRL_NC       0x0000
#define   GPCTRL_ENAB     0x4000
#define   GPCTRL_RESET    0x8000
#define   CHPTEST_NC      0x0000
```

```
#define   CHPTEST_NORMAL  0x1000
#define   CHPTEST_ENAB    0x2000
#define   IGPIDLE         0x0800
#define   IINVALIDIO      0x0400
#define   IPICKFLAG       0x0200
#define   IVBLNKFLG       0x0100
#define   RGPIDLE         0x0008
#define   RINVALIDIO      0x0004
#define   RPICKFLAG       0x0002
#define   RVBLNKFLG       0x0001

/* Current X, Y & Dest X, Y Mask */
#define   COORD_MASK      0x07FF

/* Advanced Function Control Register */
#define   CLKSEL          0x0004
#define   DISABPASSTHRU   0x0001

/* Graphics Processor Status Register */
#define   GPBUSY          0x0200
#define   DATDRDY         0x0100

/* Command Register */
#define   CMD_NOP         0x0000
#define   CMD_LINE        0x2000
#define   CMD_RECT        0x4000
#define   CMD_RECTV1      0x6000
#define   CMD_RECTV2      0x8000
#define   CMD_LINEAF      0xA000
#define   CMD_BITBLT      0xC000
#define   CMD_OP_MSK      0xF000
#define   BYTSEQ          0x1000
#define   _16BIT          0x0200
#define   PCDATA          0x0100
#define   INC_Y           0x0080
#define   YMAJAXIS        0x0040
#define   INC_X           0x0020
#define   DRAW            0x0010
#define   LINETYPE        0x0008
#define   LASTPIX         0x0004
#define   PLANAR          0x0002
#define   WRTDATA         0x0001

/*
    Short Stroke Vector Transfer Register
    (The angular defs also apply to the Command Register)
*/
#define   VECDIR_000      0x0000
#define   VECDIR_045      0x0020
#define   VECDIR_090      0x0040
#define   VECDIR_135      0x0060
```

```
#define  VECDIR_180      0x0080
#define  VECDIR_225      0x00A0
#define  VECDIR_270      0x00C0
#define  VECDIR_315      0x00E0
#define  SSVDRAW         0x0010

/* Background Mix Register */
#define  BSS_BKGDCOL     0x0000
#define  BSS_FRGDCOL     0x0020
#define  BSS_PCDATA      0x0040
#define  BSS_BITBLT      0x0060

/* Foreground Mix Register */
#define  FSS_BKGDCOL     0x0000
#define  FSS_FRGDCOL     0x0020
#define  FSS_PCDATA      0x0040
#define  FSS_BITBLT      0x0060

/*
    The Mixes (Note that some are duplicated for ease
    of use.
*/
#define  MIX_MASK                      0x001F

#define  MIX_NOT_DST                   0x0000
#define  MIX_0                         0x0001
#define  MIX_1                         0x0002

#define  MIX_DST                       0x0003
#define  MIX_LEAVE_ALONE               0x0003

#define  MIX_NOT_SRC                   0x0004

#define  MIX_SRC_XOR_DST               0x0005
#define  MIX_XOR                       0x0005

#define  MIX_NOT__SRC_XOR_DST          0x0006
#define  MIX_XNOR                      0x0006

#define  MIX_SRC                       0x0007
#define  MIX_REPLACE                   0x0007
#define  MIX_PAINT                     0x0007

#define  MIX_NOT_SRC_OR_NOT_DST        0x0008
#define  MIX_NAND                      0x0008

#define  MIX_NOT_SRC_OR_DST            0x0009
#define  MIX_SRC_OR_NOT_DST            0x000A

#define  MIX_SRC_OR_DST                0x000B
#define  MIX_OR                        0x000B
```

```
#define  MIX_SRC_AND_DST                0x000C
#define  MIX_AND                        0x000C

#define  MIX_SRC_AND_NOT_DST            0x000D
#define  MIX_NOT_SRC_AND_DST            0x000E

#define  MIX NOT SRC_AND_NOT_DST        0x000F
#define  MIX_NOR                        0x000F

#define  MIX_MIN                        0x0010
#define  MIX_DST_MINUS_SRC              0x0011
#define  MIX_SRC_MINUS_DST              0x0012
#define  MIX_PLUS                       0x0013
#define  MIX_MAX                        0x0014
#define  MIX_HALF__DST_MINUS_SRC        0x0015
#define  MIX_HALF__SRC_MINUS_DST        0x0016
#define  MIX_AVERAGE                    0x0017
#define  MIX_DST_MINUS_SRC_SAT          0x0018
#define  MIX_SRC_MINUS DST_SAT          0x001A
#define  MIX_PLUS_SAT                   0x001B
#define   MIX_HALF__DST_MINUS_SRC_SAT   0x001C
#define   MIX_HALF__SRC_MINUS_DST_SAT   0x001E
#define  MIX_AVERAGE_SAT                0x001F

/* Memory Control Register */
#define  BUFSWP          0x0010
#define  VRTCFG_2        0x0000
#define  VRTCFG_4        0x0004
#define  VRTCFG_6        0x0008
#define  VRTCFG_8        0x000C
#define  HORCFG_4        0x0000
#define  HORCFG_5        0x0001
#define  HORCFG_8        0x0002
#define  HORCFG_10       0x0003

/* Pixel Control Register */
#define   MIXSEL_FRGDMIX 0x0000
#define  MIXSEL_PATT     0x0040
#define  MIXSEL_EXPPC    0x0080
#define  MIXSEL_EXPBLT   0x00C0
#define  COLCMPOP_F      0x0000
#define  COLCMPOP_T      0x0008
#define  COLCMPOP_GE     0x0010
#define  COLCMPOP_LT     0x0018
#define  COLCMPOP_NE     0x0020
#define  COLCMPOP_EQ     0x0028
#define  COLCMPOP_LE     0x0030
#define  COLCMPOP_GT     0x0038
#define  PLANEMODE       0x0004
```

```
/*
      This makes it easier to use the palette.
*/
typedef struct
   {
   BYTE  r;
   BYTE  g;
   BYTE  b;
   } LUTENTRY;

/*
      Macro to aid in waiting for the command queue to empty.
      The parameter 'v' specifies the number of queue entries
      that you are waiting to be free.
*/
#define   WaitQueue(v)    {while (inp(GP_STAT) & (0x0100 >> (v)));}
```

Let's Start

When you use the registers and not the AI, there are many things you have to do yourself that the AI normally takes care of.

First, you need to make sure the board is present. You do this by writing a value to one of the only registers that can be both read from and written to. The register of choice is the Error Term register. We usually use the following code fragment:

```
/*
   Check to see if an 8514/A is actually installed by
   writing to the ERR_TERM register and reading back.
*/
   outpw(ERR_TERM, 0x5A5A);
   if (inpw(ERR_TERM) != 0x5A5A)
      {
       printf("No 8514/A located!!!\n");
      exit(1);
      }
```

The value 0x5A5A is arbitrary.

Unfortunately, if the 8514/A board is hung, the check will fail. The solution is to assume that an 8514/A is installed and reset it, which will get it operating normally again and will let you access the Error Term register correctly. The following code performs a reset:

```
/*Reset the 8514/A, and disable all interrupts.*/
    outpw(SUBSYS_CNTL, GPCTRL_RESET | CHPTEST_NORMAL);
    outpw(SUBSYS_CNTL, GPCTRL_ENAB  | CHPTEST_NORMAL);
```

The first line of code resets and disables the 8514/A chip set; the second line reenables it and disables all interrupt requests.

Once you have determined that the 8514/A board is present, you need to initialize the display and the drawing environment. Let's start by looking at the former.

Display Initialization

The most important thing here is to determine whether the 8514/A is capable of 4 or 8 bits per pixel (bpp) and what kind of monitor is attached. Once you know these two settings, you can safely initialize the 8514/A. This knowledge is important for two reasons.

The first is that if you initialize a 4-bpp board for 8-bpp operation, you'll end up with strange color patterns on the display. Also, in some cases, the memory-configuration registers need to be stuffed with different values for the two modes.

The second reason relates to the monitor type. If you set the sync parameters for a display mode not supported by the attached monitor, you may damage the monitor. Fortunately, this is becoming less of a concern as multifrequency monitors proliferate.

The board's pixel depth is determined by the amount of memory on the 8514/A board. The base configuration of 512K supports a 4-bpp (16-

color) depth, and when you add the 512K memory expansion (for a total of 1 Mbyte), you can display 8 bpp (256 colors). On the IBM product, you have to change a jumper position when you add the extra memory. This jumper controls the state of the 8PLANE bit in the Subsystem Status register. When this bit is set, you should be running with the full complement of memory.

The type of monitor attached to the 8514/A can also be determined by the Subsystem Status register. The 3-bit field MONITORID allows for up to eight monitor types. These three bits are produced by the monitor. This is one of the reasons why IBM started using a 15-pin connector instead of the typical 9-pin connector for the VGA when it first came out in 1987. See the description of the Subsystem Status register in Chapter 6 for a.complete list of monitor types.

While on the subject of initialization, we should explain the use of the VGA pass-through. We went into detail about how it works, but not how to control it, in Chapter 6. The VGA pass-through allows the 8514/A display to be toggled between VGA output and 8514/A output. This capability is software-controlled by the lowest bit of the Advanced Function Control register. In 1024 x 768 mode, for example, you can write a 7 to the register to enable 8514/A display information on the monitor and a 6 to enable VGA display information. The other bits set in this register control which crystal to use for display. Unfortunately, this register is not readable in the IBM implementation, so you have to make sure you know what bits are currently set before you toggle bit 0.

Listing 7-2 contains the display-initialization code for the 8514/A.

Listing 7-2: 8514/A display-initialization code

```
#define   MODE_640     0
#define   MODE_1024    1

/*--  --  -----  ----  --  ----------------------------------------------

    Init8514(res)

    Initializes the 8514/A into the requested resolution. Valid input
    values are 0 for 640x480 and 1 for 1024x768.

    No error checking on input value is performed. If an 8514/A is not
    found, or if the 1024 mode is selected, but an appropriate monitor
    is not available, this routine will print a message and exit to DOS.

-----  ---  ---------  -------  ---  ---------  --------  -  -----  ---------*/
VOID   Init8514(res)
WORD   res;
{
    static    WORD   initRegs[] =
                     {
                      DISP_CNTL, ADVFUNC_CNTL, V_TOTAL, V_SYNC_STRT,
                      V_DISP, H_TOTAL, H_SYNC_STRT, H_DISP,
                      H_SYNC_WID, V_SYNC_WID, DISP_CNTL, 0
                     };
    static    WORD   mode640x4[] =
                     {
                      0x0041, 0x0003, 0x0830, 0x07A8,
                      0x0779, 0x0063, 0x0052, 0x004F,
                      0x002C, 0x0022, 0x0021
                     };
    static    WORD   mode640x8[] =
                     {
                      0x0043, 0x0003, 0x0418, 0x03D2,
                      0x03BB, 0x0063, 0x0052, 0x004F,
                      0x002C, 0x0022, 0x0023
                     };
    static    WORD   mode1024x4[] =
                     {
                      0x0053, 0x0007, 0x0660, 0x0600,
                      0x05FB, 0x009D, 0x0081, 0x007F,
                      0x0016, 0x0008, 0x0033
                     };
    static    WORD   mode1024x8[] =
                     {
                      0x0053, 0x0007, 0x0660, 0x0600,
                      0x05FB, 0x009D, 0x0081, 0x007F,
                      0x0016, 0x0008, 0x0033
                     };
```

```
static    WORD  *modeList[] =
                  {
                    mode640x4, mode1024x4, mode640x8, mode1024x8
                  };
static    WORD  numPlanes;


WORD  i;

/*
    Blank the screen temporarily to display color 0 by turning
    the display of all planes off.
*/
 outp(DAC_MASK, 0x00);

/*
    Reset the 8514/A, and disable all interrupts.
*/
 outpw(SUBSYS_CNTL, GPCTRL_RESET | CHPTEST_NORMAL);
 outpw(SUBSYS_CNTL, GPCTRL_ENAB | CHPTEST_NORMAL);

/*
    Check to see if an 8514/A is actually installed by
    writing to the ERR_TERM register, and reading back. The
    0x5A5A value is entirely arbitrary.
*/
 outpw(ERR_TERM, 0x5A5A);
 if (inpw(ERR_TERM) != 0x5A5A)
    {
     printf("No 8514/A located!!!\n");
     exit(1);
    }

/*
    Get board status information.
*/
 i = inpw(SUBSYS_STAT);

/*
    Check if monitor capable of 1024x768 is attached. If
    not, print error message and abort. In this case, we
    want the IBM 8514 monitor.
*/
 if ((i & MONITORID_MASK) != MONITORID_8514)
    {
     printf("Attached monitor not capable of color 1024x768!!!");
     exit(2);
    }

/*
    Determine the pixel depth of the board. Then, based
```

```
        on depth, set the memory control register to use the
        appropriate CAS (VRAM Column Address Select) configuration.
    */
    if (i & _8PLANE)
        {
        numPlanes = 8;
        outpw(MULTIFUNC_CNTL, MEM_CNTL | VRTCFG_4 | HORCFG_8);
        }
    else
        {
        numPlanes = 4;
        if (res == MODE_640)
            outpw(MULTIFUNC_CNTL, MEM_CNTL | VRTCFG_2 | HORCFG_8);
        else
            outpw(MULTIFUNC_CNTL, MEM_CNTL | VRTCFG_4 | HORCFG_8);
        }

    /*
        Now initialize the display controller part of the 8514/A.
        We have set up the initialization values in two sets of
        tables: 4 bpp and 8 bpp. The "numPlanes" comparison below
        determines which set of values to use (dual indices).
    */
    i = 0;
    while (initRegs[i])
        {
        outpw(initRegs[i], modeList[res + (numPlanes == 4 ? 0:2)][i]);
        i++;
        }

    /*
        Now reenable the screen, but only the planes that actually exist.
        Otherwise, you end up with bus noise on the display.
    */
    if (numPlanes == 8)
        outpw(DAC_MASK, 0xFF);              /* Enable all 8 planes for display */
    else
        outpw(DAC_MASK, 0x0F);                /* Enable only lower 4 planes    */

    return;
}
```

Drawing Environment

After a reset, the 8514/A's state is uncertain, so it is best to set things to the state you are most interested in. In our programming examples, we initialize the drawing environment in the following manner:

Current drawing position	(0, 0)
Foreground mix	Dst = Src; use foreground color for source
Background mix	Dst = Src; use background color for source
Foreground color	0xFFFF
Background color	0
Clipping rectangle	Set to allow drawing in drawable space
Write mask	Enable all planes for update
Color-compare operation	FALSE (compares don't affect updates)
Color palette	See Figure 5-2

All other parameters are left uninitialized.

A description of the operational environment is given at the beginning of Chapter 5.

Mentioning the current drawing position and the clipping rectangle brings us to the coordinate system. As with the AI, the 8514/A uses an (x, y) coordinate system with the origin in the upper left corner of the display. Table 4-1 shows the limits of the coordinate system.

There is one exception to the normal (x, y) addressing, and this is the 640 x 480 by 16-color mode. In this mode, the IBM 8514/A's memory architecture is used somewhat differently, requiring the desired y coordinate of any operation to be translated to one understood by the hardware. Because this particular mode is the least used of the 8514/A modes, we don't provide any programming examples to show you how to use it. Just in case you need to, though, the following bit of code shows you how to perform the y-coordinate translation:

```
WORD   TransY(y)
WORD   y;
{
return((y & 1) | ((y >> 1) << 2));
}
```

Listing 7-3 contains the code to set up the drawing environment. **Note that we assume the 8514/A is in 1024 x 768-resolution mode. This assumption applies to all subsequent programs in this chapter as well.**

Listing 7-3: Initializing the 8514/A drawing environment

```
/* ------------------------------------------------------------------ -----

    InitLUT()

    Loads the Look-up Table with ramps of red, green, blue, and gray.
    Assumes 8-bit board is in use. If 4 bit board, then only the first
    16 entries in LUT will be used. Could be written more compact, but
    has been expanded for greater readability.

------------------------------------------------------------------ ------*/
VOID   InitLUT()
{
    static LUTENTRY lutData[16] =          /* LUT Init designed to approxi- */
                {                           /*  mate default VGA colors.     */
                0x00, 0x00, 0x00,          /* Black                         */
                0x00, 0x00, 0x7F,          /* Dark Blue                     */
                0x00, 0x7F, 0x00,          /* Dark Green                    */
                0x00, 0x7F, 0x7F,          /* Dark Cyan                     */
                0x7F, 0x00, 0x00,          /* Dark Red                      */
                0x7F, 0x00, 0x7F,          /* Dark Magenta                  */
                0x7F, 0x7F, 0x00,          /* Dark Yellow (Brown)           */
                0x7F, 0x7F, 0x7F,          /* Gray                          */
                0x1F, 0x1F, 0x1F,          /* Dark Gray (Almost Black)      */
                0x00, 0x00, 0xFF,          /* Bright Blue                   */
                0x00, 0xFF, 0x00,          /* Bright Green                  */
                0x00, 0xFF, 0xFF,          /* Bright Cyan                   */
                0xFF, 0x00, 0x00,          /* Bright Red                    */
                0xFF, 0x00, 0xFF,          /* Bright Magenta                */
                0xFF, 0xFF, 0x00,          /* Bright Yellow                 */
                0xFF, 0xFF, 0xFF           /* White                         */
                };
```

```
    WORD      i, j;

    /*
        Initialize the first 16 entries.
        Start by setting the DAC Write Index to 0.
    */
     outp(DAC_W_INDEX, 0);

    /*
        Load the first 16 LUT entries.
    */
    for (i = 0; i < 16; i++)
       {
       /*
            Write out each lut entry's red, green, and blue portions.
            Since the 8514/A DAC only has 6 bits of precision, we have
            to shift the data right.
       */
        outp(DAC_DATA, lutData[i].r >> 2);
        outp(DAC_DATA, lutData[i].g >> 2);
        outp(DAC_DATA, lutData[i].b >> 2);
       }

    /*
        Load the remaining 240 LUT entries (15 sets of 16).
    */
    for (i = 1; i < 16; i++)
       for (j = 0; j < 16; j++)
          {
           outp(DAC_DATA, lutData[i].r >> 2);
           outp(DAC_DATA, lutData[i].g >> 2);
           outp(DAC_DATA, lutData[i].b >> 2);
          }

    return;
}


/*-----  --  --  -- --  ----  -- -  - -- ----- ----------- --------

    InitEnvironment()

    Initializes the 8514/A's drawing environment and clears the display.

---- --- ---- -- -- -  --  - --- --  ---- ------- ----- --*/
VOID   InitEnvironment()
{
    /*
        Current mixes, replace, foreground active
    */
     outpw(FRGD_MIX, FSS_FRGDCOL | MIX_REPLACE);
```

```
    outpw(BKGD_MIX, BSS_BKGDCOL | MIX_REPLACE);

    /*
        Clipping rectangle to full drawable space (0,0 to 1023, 1023).
    */
    outpw(MULTIFUNC_CNTL, SCISSORS_T | 0x000);
    outpw(MULTIFUNC_CNTL, SCISSORS_L | 0x000);
    outpw(MULTIFUNC_CNTL, SCISSORS_B | 0x3FF);
    outpw(MULTIFUNC_CNTL, SCISSORS_R | 0x3FF);

    /*
        Enable writes to all planes and reset color compare
    */
    outpw(WRT_MASK, 0xFFFF);
    outpw(MULTIFUNC_CNTL, PIX_CNTL | 0x0000);       /* Clear lower bits */

    /*
        Clear the display. Need to set the color, origin, and size. Then draw.
        See Listing 7-5 for more information.
    */
    WaitQueue(6);
    outpw(FRGD_COLOR, 0);
    outpw(CUR_X, 0);
    outpw(CUR_Y, 0);
    outpw(MAJ_AXIS_PCNT, 1023);
    outpw(MULTIFUNC_CNTL, MIN_AXIS_PCNT | 1023);
    outpw(CMD, CMD_RECT | INC_Y | INC_X | DRAW | PLANAR | WRTDATA);
    WaitQueue(4);

    /*
        Reset current draw position
    */
    outpw(CUR_X, 0x0000);
    outpw(CUR_Y, 0x0000);

    /*
        Reset current colors, foreground is all on, background is 0.
    */
    outpw(FRGD_COLOR, 0xFFFF);
    outpw(BKGD_COLOR, 0x0000);

    /*
        Load the Look-Up Table
    */
    InitLUT();

    return;
}
```

Down to the Nitty-Gritty

Now that we have covered initialization, look at the program in Listing 7-4. It ties together the board check, display initialization, and environment initialization in a shell that calls the routine **DoSomething()**. (This shell is the register version of the AI program shell in Listing 5-4.)

Writing code for the 8514/A register set, although tedious at times, is relatively simple. You need to deal with only a handful of key registers once you have initialized the hardware. Of these, the two most used registers are the Graphics Processor Status register and the Command register. Refer to Chapter 6 for a detailed explanation of these registers.

As you will see, applications use the Graphics Processor Status register to determine how many slots are available in the 8514/A's command queue. Commands that are queued include the setting of line-drawing parameters, colors, and mixes, as well as the actual Command register drawing instructions.

Listing 7-4: 8514/A program shell

```
/*
    LIST7-4.C - Shell to initialize the 8514/A
                Written by Jake Richter
                 Copyright (c) 1989,1990  Panacea Inc., Londonderry, NH

                 This program may be freely distributed and modified.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <dos.h>
#include <conio.h>
#include <io.h>
#include <malloc.h>
#include <sys\types.h>
```

```
#include <sys\stat.h>
#include "reg8514.h"                        /* Located in LIST7-1.H          */

/*
    Include code from Listings 7-2 and 7-3 here.
*/
#include "LIST7-2.C"
#include "LIST7-3.C"



/*
    This is where you should include the code from the other
    listings, one at a time, in order to test and examine how
    the code works.
*/



/*
    The program entry point.
*/
main()
{
    Init8514(MODE_1024);                    /* Initialize the 1024x768 mode  */

    InitEnvironment();                      /* Init the drawing environment  */

    DoSomething();                          /* Do our graphics stuff here!   */

    getch();                                /* Wait for it to be seen.       */

    outpw(ADVFUNC_CNTL, 6);                 /* Return to VGA Mode            */

    return(0);
}
```

Clearing the Display

Once you have properly initialized a graphics board, the first real drawing operation you usually perform is to to clear the display (that is, clear all display memory). With the 8514/A, the simplest way to do this is to draw a filled rectangle in color 0. We have therefore provided a general-purpose rectangle-filling routine in Listing 7-5. To clear the display, you just need to specify the origin (0, 0) and extents of the display or display buffer ([1024, 768] or [1024, 1024], respec-

tively), as well as a drawing color, and voila, the display will be cleared to the color you chose. **Notice that the Major and Minor Pixel Count registers are actually the number of pixels, less one**. This is an important fact to remember when you deal with most of the drawing functions.

Listing 7-5: Filling rectangles using the 8514/A registers

```
/*------------------------------ ---------------                ----------

    FillRectangle(x, y, width, height, color)

    Draws a filled rectangle at point x,y with dimensions
    width * height. Existing mixes, clipping, etc. apply.
    It is assumed that width and height are positive.
    No error checking on values is performed.

-------------------------------   - -- - ------- -- - --- - - --  */
VOID   FillRectangle(x, y, width, height, color)
WORD   x, y, width, height, color;
{
   /*
       Wait for six entries to free up.
   */
    WaitQueue(6);

   /*
      Set color
   */
    outpw(FRGD_COLOR, color);

   /*
      Set start point
   */
    outpw(CUR_X, x);
    outpw(CUR_Y, y);

   /*
       Set height and width. Note that the 8514/A actually uses
       deltas, not pixel counts.
   */
    outpw(MAJ_AXIS_PCNT, width - 1);
    outpw(MULTIFUNC_CNTL, MIN_AXIS_PCNT | (height - 1));

   /*
      Fill the Rectangle
```

```
    */
    outpw(CMD, CMD_RECT | INC_Y | INC_X | DRAW | PLANAR | WRTDATA);

    return;
}


/*--------------------------- ---------- - ----- ------ ------ -------

    DoSomething()

    Draw a series of rectangles of various colors using the FillRectangle()
    routine listed above. Called from LIST7-4.C

-------------------------------------------------------------------------*/
VOID   DoSomething()
{
    WORD  i;

    for (i = 0; i < 500; i ++)              /* Then draw lots of rectangles  */
        FillRectangle(i, i, i + 10, i + 8, i & 0x0F);

    return;
}
```

Line Drawing

Line drawing is probably the most used capability of graphics boards these days, and the 8514/A would be incomplete without it. Unfortunately, the designers of the 8514/A did not go far enough with their implementation. Line drawing on the 8514/A requires substantial setup, thereby limiting its efficiency. On a positive note, some of the 8514/A-clone companies have come up with a simplified mechanism that greatly speeds the drawing of lines.

The setup for lines involves calculating three special values—namely, the axial step constant, the diagonal step constant, and the line error term. These values are an integral part of the popular Bresenham line-drawing algorithm used by the 8514/A. The code sample in Listing 7-6 shows how to generate the appropriate parameters and draw a line with them.

Listing 7-6: Drawing lines using the 8514/A registers

```
/* ------------------- ---- --- -- --- --  ---------- --- ---- - -----

    DrawLine(x1, y1, x2, y2, color)

    Draws a line from point x1,y1 to point x2,y2 in using color.
    Existing mixes, clipping, etc. apply. No error checking on
    values is performed.

--------- -- ---------- --- ---- ----  --------------- ------------*/
VOID   DrawLine(x1, y1, x2, y2, color)
WORD   x1, y1, x2, y2, color;
{
    WORD   command = CMD_LINE | DRAW | PLANAR | WRTDATA;
    WORD   deltaX, deltaY;
    WORD   fixup = 0;
    WORD   majorAxis, minorAxis;
    WORD   twiceMinor;


    /*
        Calculate the distance between points for each axis
    */
    deltaX = x1 - x2;
    deltaY = y1 - y2;


    if (deltaX < 0)
       {
       /*
           If X2 > X1, make delta positive and set
           the INC_X bit in command
       */
       deltaX = -deltaX;
       command |= INC_X;
       fixup = -1;
       }

    if (deltaY < 0)
       {
       /*
           If Y2 > Y1, make delta positive and set
           the INC_Y bit in command
       */
       deltaY = -deltaY;
       command |= INC_Y;
       }

    if (deltaX < deltaY)
```

```
    {
    /*
        If Y delta is bigger, set the MAJAXIS bit in command.
    */
     command |= YMAJAXIS;
     majorAxis = deltaY;
     minorAxis = deltaX;
    }
else
    {
    /*
        If X delta is bigger...
    */
     majorAxis = deltaX;
     minorAxis = deltaY;
    }

/*
    Precalculate, as this is used repeatedly.
*/
 twiceMinor = minorAxis * 2;

/*
    Wait for eight entries in queue to be available
*/
 WaitQueue(8);

/*
    Set the foreground color
*/
 outpw(FRGD_COLOR, color);

/*
    Set the line start position
*/
 outpw(CUR_X, x1);
 outpw(CUR_Y, y1);

/*
    Set the line drawing parameters
*/
 outpw(ERR_TERM, twiceMinor - majorAxis + fixup);
 outpw(DESTY_AXSTP, twiceMinor);
 outpw(DESTX_DIASTP, twiceMinor - 2 * majorAxis);
 outpw(MAJ_AXIS_PCNT, majorAxis);
/*
    Write the command to the 8514/A
*/
 outpw(CMD, command);

 return;
```

```
}

/*-----------------------------------------------------------------------

    DoSomething()

    Draw a sweep of lines of various colors using the DrawLine()
    routine listed above. Called from LIST7-4.C

-----------------------------------------------------------------------*/
VOID   DoSomething()
{
   WORD  i;

   for (i = 0; i < 500; i ++)
      DrawLine(0, 0, 800, i, i & 0x0F);

   return;
}
```

Long Vectors

If you only need to draw lines that are horizontal, vertical, or purely diagonal (multiples of 45 degrees), you may want to use long vectors. This is an extremely efficient way to draw lines quickly because it requires you to set only two registers — the Major Axis Pixel Count register and the Command register. Normal line drawing, as you have seen, requires that four registers be written to, and those only after some calculations. A sample routine that draws long vectors is provided in Listing 7-7.

Listing 7-7: Drawing long vectors using the 8514/A registers

```
/*---------------------------- ----------------------- ---------------

    LongVector(x, y, length, direction, color)

    Draws a long vector starting at x,y in the direction specified. The
    vector consists of "length" pixels, using color.
    Existing mixes, clipping, etc. apply. No error checking on
    values is performed.

    The direction is a number ranging from 0 to 7, and indicates a
    counter-clockwise degree direction. The angle is calculated by
    multiplying 45 by the direction value.

-------------------------------------------------------------------------*/
VOID   LongVector(x, y, length, direction, color)
WORD   x, y, length, direction, color;
{
   /*
       Wait for eight entries in queue to be available
   */
    WaitQueue(5);

   /*
       Set the foreground color
   */
    outpw(FRGD_COLOR, color);

   /*
       Set the line start position
   */
    outpw(CUR_X, x);
    outpw(CUR_Y, y);

   /*
       Set the length of the vector.
   */
    outpw(MAJ_AXIS_PCNT, length - 1);
   /*
       Write the command to the 8514/A, making sure to shift the direction
       into place.
   */
    outpw(CMD, CMD_LINE | DRAW | LINETYPE | PLANAR
                 | WRTDATA | ((direction & 0x0007) << 5));

    return;
}
```

```
/*--------------- -------------------- ----------------------------

    DoSomething()

    Draw a series of lines of various colors using the LongVector()
    routine listed above. Called from LIST7-4.C

---------------------------- ------------------------------------------*/
VOID   DoSomething()
{
   WORD  i, j;

   for (i = 300; i > 0; i -= 10)
      for (j = 0; j < 8; j++)
         LongVector(512, 384, i, j, (i / 10) & 0x0F);

   return;
}
```

Short-Stroke Vectors

Another form of line drawing that is fast, but limited in application, is to use short-stroke vectors, a subset of long vectors. Short-stroke vectors are limited to at most 15 pixels in length and can be drawn only at angles that are multiples of 45 degrees. IBM uses this capability of the 8514/A to draw text. Short-stroke vectors can consist of either a draw or a move action, thereby providing great flexibility. For more information about short-stroke vectors, see the description of the Short Stroke Vector Transfer register in Chapter 6.

Because the IBM 8514/A board comes with short-stroke-vector fonts, we have provided an example in Listing 7-8 that loads a font and then draws characters on the display. Modification of this routine to support single- and multiplane bit-mapped fonts is left as an exercise.

Listing 7-8:- Loading and using a short-stroke-vector font

```
/*
    Font file header
*/
typedef struct CharSetDef                    /* Character Set Definition block */
    {
    UBYTE     reserve1;
    UBYTE     type;
    DWORD     reserve2;
    UBYTE     reserve3;
    UBYTE     cellwidth;
    UBYTE     cellheight;
    UBYTE     reserve4;
    WORD      cellnbytes;
    WORD      flags;
    WORD FAR *indextbl;
    UBYTE FAR *enveltbl;
    UBYTE     cdpt1st;
    UBYTE     cdptlast;
    UBYTE FAR *chardef1;
    WORD      reserve5;
    UBYTE FAR *chardef2;
    WORD      reserve6;
    UBYTE FAR *chardef3;
    }  CHARSETDEF;


typedef struct FontFileDefn
    {
    WORD  no_pages; /* no of code pages in the file (size of page_array) */
    WORD  def_page; /* default page index (into page_array)              */
    WORD  alt_page; /* alternate default page index (into page_array)    */
    struct
       {
       BYTE  code_page_id[4];
       WORD  csd_offset;
       } page_array[1];
    } FFDEF;


static   FFDEF       *ioAddr;              /* Address to load font to       */

/*  -- --- -- --  -  ---  -  --  -  -- -- -- --

    LoadFont(name)

    This routine loads fonts from the files supplied by IBM with the 8514/A.
    The function requires the file name of the AI compatible font file, and
    if the font is successfully loaded, returns a pointer to the character
```

```
    set definition, as required by the AI. If the return value is 0, an
    error occurred while trying to load the file. Also, if you really want
    to be neat about things, the memory allocated for the font should be
    freed prior to exiting to DOS. Refer to Appendix C for more information
    on font files.

------------------------------ ---------------------------------------- */
CHARSETDEF   *LoadFont(name)
BYTE   *name;
{
          UWORD        fontLen;              /* Size of font to load          */
          WORD         fontHdl;              /* Font file handle              */
          CHARSETDEF   *pCSD;                /* Pointer to Char Set Def.      */

                                             /* Open the Font file            */
    if ((fontHdl = open(name, O_RDONLY|O_BINARY)) == -1)
       {
        printf("Could not open font file '%s'!!!\n", name);
        return(0);
       }
                                             /* Get size of font file         */
    fontLen = (UWORD) lseek(fontHdl, 0L, SEEK_END);

    ioAddr = (FFDEF *) malloc(fontLen);      /* Allocate enough space for file*/
    if (!ioAddr)
       {
        printf("Not enough memory to load font!\n");
        return(0);
       }

    lseek(fontHdl, 0L, SEEK_SET);            /* Return to start of file       */
    read(fontHdl, (BYTE *)ioAddr, fontLen);/* Load the whole file into mem. */

                                              /* Point to start of def. font   */
    pCSD = (CHARSETDEF *) (((BYTE *) ioAddr) +
               ioAddr->page_array[ioAddr->def_page].csd_offset);

                               /* Update all address fields with load address */
    pCSD->chardef1 = ((UBYTE FAR *) ioAddr) + ((DWORD) pCSD->chardef1);
    pCSD->chardef2 = ((UBYTE FAR *) ioAddr) + ((DWORD) pCSD->chardef2);
    pCSD->chardef3 = ((UBYTE FAR *) ioAddr) + ((DWORD) pCSD->chardef3);
    pCSD->indextbl = (WORD FAR *)
                        (((UBYTE FAR *) ioAddr) + ((DWORD) pCSD->indextbl));
    pCSD->enveltbl = ((UBYTE FAR *) ioAddr) + ((DWORD) pCSD->enveltbl);


    close(fontHdl);                          /* Close the font handle         */

    return(pCSD);                            /* Return pointer to the font    */
}
```

```
/* --  ------- -- - -- ------------------------------------ ------------------

    DrawCharacter(x, y, character, color, fontPtr)

    Draws a short stroke vector character from the font defined by fontPtr.
    Returns without doing anything if font type isn't short stroke.
    Remember that with the IBM version of the font, x,y specify the lower
    left corner of the font cell. In a really well-behaved program, you
    should also check to see if the character being passed is a valid one
    for the current font definition. Current mixes, clipping rectangle, etc.
    are observed:

---------- ---- ------   ---------- --  -- ---- ----  ---- --    --- --- - */
VOID   DrawCharacter(x, y, character, color, fontPtr)
WORD         x, y;
UWORD         character, color;
CHARSETDEF   *fontPtr;
{
    register UWORD        vecs;
             UWORD FAR    *charPtr;

    if (fontPtr->type != 3)
       return;                                  /* Return if not SSV font        */

                                                /* Point to the char definition  */
     charPtr = (WORD FAR *)(fontPtr->chardef1 + fontPtr->indextbl[character]);

    while (inpw(GP_STAT) & BUSY);               /* Wait for existing command to  */
                                                /*  clear                        */
    WaitQueue(4);                               /* Make sure we have queue space */

    outpw(FRGD_COLOR, color);                  /* Set the color                 */

    outpw(CUR_X, x);                            /* Starting point                */
     outpw(CUR_Y, y);
                                                /* Set the command register for  */
                                                /*  short stroke vectors         */
     outpw(CMD, CMD_NOP | BYTSEQ | LINETYPE | LASTPIX | PLANAR | WRTDATA);
    outpw(SHORT_STROKE, 0);                     /* Also clear the SSV buffer     */

    do
       {
       vecs = *charPtr++;                       /* Get a SSV word                */
       if (!(vecs & 0x00FF))                    /* If first vec is nul, we's done*/
          break;
       outpw(SHORT_STROKE, vecs);               /* Send it out to the board      */
       } while (vecs & 0xFF00);                 /* Quit if 2nd vec is nul        */

    return;
}
```

```
/*----  ------------------------- -------  ------------  --------------

    DoSomething()

    Print a string of characters using LoadFont and DrawCharacter.

------------------------------------------------------------------------------*/
VOID   DoSomething()
{
   static    BYTE          string[] = "Hello World!";
             BYTE          *strPtr = string;
             WORD          x = 100, y = 100;
             CHARSETDEF    *fontPtr;


                                          /* Load the font - makes an       */
                                          /*  assumption about file name    */
    fontPtr = LoadFont("\\hdipcdos\\stan1220.fnt");
   if (!fontPtr)                          /* Return if no font loaded       */
      return;

   /*
       NOTE: Short stroke fonts, which the STAN????.FNT files happen to
       contain, do not clear the background when displayed. This means that
       you need to do a filled rectangle to clear an area before displaying
       short stroke font characters. This can have some positive uses as well,
       especially if you don't want to alter the background of the character
      cell.
   */
   while (*strPtr)                       /* Print the string               */
      {                                  /* Draw each character            */
       DrawCharacter(x, y, *strPtr++, 0xFF, fontPtr);
      x += fontPtr->cellwidth;           /* Update position                */
      }

   free(ioAddr);                          /* Free up the font buffer.       */
    return;
}
```

BitBLTs

One of the strong areas of the 8514/A is its fast BitBLT capability. BitBLTs let you move rectangular regions of image data rapidly from one place to another. BitBLTs on the 8514/A can consist of pixels, planes, system data, or a combination thereof. Listing 7-9 gives an ex-

ample of how to do an all-plane BitBLT. Note that you have to change the foreground mix source in order to perform a BitBLT.

Single-plane expansion is another useful function of BitBLTs. The purpose of a single-plane expansion is to take monochromatic data and expand it into multiplane (color) data. This applies to bit-mapped fonts, masks, and other similar types of image sources. With the 8514/A, you specify a plane to be used as the source for the expansion. All 0 bits in the source area indicate that the background mix should be used (which determines the background source select and mix mode), and all 1 bits select the foreground mix. Normally the expansion is used for expanding all 0s and 1s to background and foreground colors, respectively, but by playing with the mixes and sources, you can do some pretty interesting things. You can derive the code to do this by examining the setup in Listing 7-11 (see the following section on image transfers). Note that you must set the MIXSEL field in the Pixel Control register in order to set this mode.

Listing 7-9: An all-plane BitBLT

```
/*      ----- ---- -        ------------ ----   ----  ----   --------  --

    FillRectangle(x, y, width, height, color)

    Draws a filled rectangle at point x,y with dimensions
    width * height. Existing mixes, clipping, etc. apply.
    It is assumed that width and height are positive.
    No error checking on values is performed.

      ---  ---  -          -  --  --  ---   ----  ---------  ----------*/
VOID   FillRectangle(x, y, width, height, color)
WORD   x, y, width, height, color;
{
   /*
       Wait for six entries to free up.
   */
    WaitQueue(6);

   /*
       Set color
   */
```

```
    outpw(FRGD_COLOR, color);

    /*
        Set start point
    */
    outpw(CUR_X, x);
    outpw(CUR_Y, y);

    /*
        Set height and width. Note that the 8514/A actually uses
        deltas, not pixel counts.
    */
    outpw(MAJ_AXIS_PCNT, width - 1);
    outpw(MULTIFUNC_CNTL, (height - 1) | MIN_AXIS_PCNT);

    /*
        Fill the Rectangle
    */
    outpw(CMD, CMD_RECT | INC_Y | INC_X | DRAW | PLANAR | WRTDATA);

    return;
}
/*--------------------------------------------------------------------

    BitBLT(srcX, srcY, dstX, dstY, width, height)

    Copies a width by height portion of display memory from
    the source location to the destination in replace mode.
    Existing clipping, color compare, and plane enables apply, but mixes
    don't (only because of how we coded this).

    NOTE: It is assumed that width and height are positive. For overlapping
    BitBLTs, you have to adjust the start position and the directional
    indicators (INC_X and INC_Y). This is left as an exercise.

    No error checking on values is performed. The foreground and background
    mix are modified.

------------------------------ ------- --------- ---------------    . */
VOID   BitBLT(srcX, srcY, dstX, dstY, width, height)
WORD   srcX, srcY, dstX, dstY, width, height;
{

    /*
        Wait for eight entries to free up.
    */
    WaitQueue(8);

    /*
        Set start point
    */
```

```
    outpw(CUR_X, srcX);
    outpw(CUR_Y, srcY);

    /*
        Set destination point
    */
    outpw(DESTX_DIASTP, dstX);
    outpw(DESTY_AXSTP, dstY);

    /*
        Set height and width. Note that the 8514/A actually uses
        deltas, not pixel counts.
    -*/
    outpw(MAJ_AXIS_PCNT, width - 1);
    outpw(MULTIFUNC_CNTL, (height - 1) | MIN_AXIS_PCNT);

    /*
        Set special Mixes for blit replace mode
    */
    outpw(FRGD_MIX, FSS_BITBLT | MIX_REPLACE);
    outpw(BKGD_MIX, BSS_BKGDCOL | MIX_REPLACE);

    /*
        Wait for three more entries to free up.
    */
    WaitQueue(3);

    /*
        Copy the Rectangle
    */
    outpw(CMD, CMD_BITBLT | INC_Y | INC_X | DRAW | PLANAR | WRTDATA);

    /*
        Reset mixes to replace mode
    */
    outpw(FRGD_MIX, FSS_FRGDCOL | MIX_REPLACE);
    outpw(BKGD_MIX, BSS_BKGDCOL | MIX_REPLACE);

    return;
}


/*  --- -   --  --  - -  --  - --  --  -- --- ---  -- -- ------

    DoSomething()

    Draw a 4x4 block of rectangles of various colors using the FillRectangle()
    routine from Listing 7-5, and then BitBLT the result around the display.

----------  - -  ------  ------- --  --------- --------------------*/
VOID  DoSomething()
```

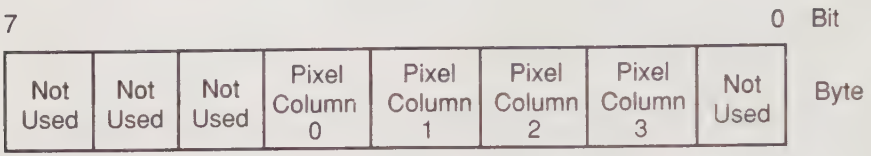
```
{
    WORD  i, j;

    for (i = 0; i < 16; i++)
        FillRectangle((i & 3) * 20, (i >> 2) * 20, 20, 20, i);

    for (i = 0; i < 16; i++)
        BitBLT(0, 0, (i & 3) * 80, (i >> 2) * 80 + 100, 80, 80);

    return;
}
```

Image Transfers

Using the same sort of approach as you would with a BitBLT, you can load images to the 8514/A board. The only way this differs from a regular BitBLT is that the foreground mix source has to be set to PC data and the PCDATA flag has to be set in the Command register. Of course, you can use the rectangle command or even horizontal vector draw as well. We recommend the latter approach.

Sending monochrome information up the board is equally important, especially when dealing with bit-mapped fonts. The principle here is similar to that of the single-plane expansion that occurs entirely on-board the 8514/A. There is one major exception, though, and that is that monochrome data transfers to the board are nugget-oriented. This means that monochrome data has to be sent to the board four bits at a time, and to make matters worse, that data has to be in bit positions 4, 3, 2, and 1 (see Figure 7-1). Bit 4 corresponds to pixel column 0 VRAMs, which translates to being every fourth pixel position starting at $x = 0$. Ergo, pixel column 0 is pixels 0, 4, 8, 12, . . .; bit 3 corresponds to pixel column 1 (pixels 1, 5, 9, . . .); bit 2 to column 2; and bit 1 to column 3. Therefore, you must also shift the source data to properly handle starting BLT positions that are not multiples of 4. This problem gets more complicated at a horizontal resolution of 1280 because you then have to contend with pixel column 4 and five bits per nugget.

Figure 7-1: Monochrome data transfer format



Listings 7-10 and 7-11 contain the only assembly-language code in this book. Listing 7-10 is the assembly-language version of the register definitions in Listing 7-1 (along with a couple of useful macros), and Listing 7-11 is a subroutine to do the monochrome expansion from PC memory to the 8514/A. We have supplied these in assembly language because supporting the register rotations in C is too cumbersome.

Note that we use a special trick that uses the left and right clipping edges to eliminate pixel relics when not expanding into full nuggets. Also, because of the 8514/A's addressing mechanism, the Background Color and Pixel Data Transfer registers interact, which means you must reinitialize the Background Color register after every completed pixel data-transfer operation. Of course, if the expansion you are performing doesn't require the background color due to the background source select or mix, then you don't need to reset it. Listing 7-12 provides some C code that demonstrates the use of the assembly-language monochrome expansion routines.

Listing 7-10: The assembly-language include file for the 8514/A

```
;
;
;       REG8514.INC
;
;       Written by Jake Richter
;       Copyright (c) 1989, 1990 Panacea Inc., Londonderry, NH
;
;       This code may be freely incorporated in any program without
;       royalty, as long as the copyright notice stays intact.
;
```

```
;       This include file contains the standard definitions
;       necessary for programming the IBM 8514/A.
;
;
;
;       This is a list of all registers and bit fields.
;
DAC_MASK          equ    02EAH
DAC_R_INDEX       equ    02EBH
DAC_W_INDEX       equ    02ECH
DAC_DATA          equ    02EDH
DISP_STAT         equ    02E8H
H_TOTAL           equ    02E8H
H_DISP            equ    06E8H
H_SYNC_STRT       equ    0AE8H
H_SYNC_WID        equ    0EE8H
V_TOTAL           equ    12E8H
V_DISP            equ    16E8H
V_SYNC_STRT       equ    1AE8H
V_SYNC_WID        equ    1EE8H
DISP_CNTL         equ    22E8H
ADVFUNC_CNTL      equ    4AE8H
SUBSYS_STAT       equ    42E8H
SUBSYS_CNTL       equ    42E8H
ROM_PAGE_SEL      equ    46E8H
CUR_Y             equ    82E8H
CUR_X             equ    86E8H
DESTY_AXSTP       equ    8AE8H
DESTX_DIASTP      equ    8EE8H
ERR_TERM          equ    92E8H
MAJ_AXIS_PCNT     equ    96E8H
GP_STAT           equ    9AE8H
CMD               equ    9AE8H
SHORT_STROKE      equ    9EE8H
BKGD_COLOR        equ    A2E8H
FRGD_COLOR        equ    A6E8H
WRT_MASK          equ    AAE8H
RD_MASK           equ    AEE8H
COLOR_CMP         equ    B2E8H
BKGD_MIX          equ    B6E8H
FRGD_MIX          equ    BAE8H
MULTIFUNC_CNTL    equ    BEE8H
MIN_AXIS_PCNT     equ    0000H
SCISSORS_T        equ    1000H
SCISSORS_L        equ    2000H
SCISSORS_B        equ    3000H
SCISSORS_R        equ    4000H
MEM_CNTL          equ    5000H
PATTERN_L         equ    8000H
PATTERN_H         equ    9000H
PIX_CNTL          equ    A000H
```

```
PIX_TRANS       equ   0E2E8H
;
; Display Status Bit Fields
;
HORTOG          equ   00004H
VBLANK          equ   00002H
SENSE           equ   00001H
;
; Horizontal Sync Width Bit Field
;
HSYNCPOL_NEG    equ   00020H
HSYNCPOL_POS    equ   00000H
;
; Vertical Sync Width Bit Field
;
VSYNCPOL_NEG    equ   00020H
VSYNCPOL_POS    equ   00000H
;
; Display Control Bit Field
;
DISPEN_NC       equ   00000H
DISPEN_DISAB    equ   00040H
DISPEN_ENAB     equ   00020H
INTERLACE       equ   00010H
DBLSCAN         equ   00008H
MEMCFG_2        equ   00000H
MEMCFG_4        equ   00002H
MEMCFG_6        equ   00004H
MEMCFG_8        equ   00006H
ODDBNKENAB      equ   00001H
;
; Subsystem Status Register
;
_8PLANE         equ   00080H
MONITORID_8503  equ   00050H
MONITORID_8507  equ   00010H
MONITORID_8512  equ   00060H
MONITORID_8513  equ   00060H
MONITORID_8514  equ   00020H
MONITORID_NONE  equ   00070H
MONITORID_MASK  equ   00070H
GPIDLE          equ   00008H
INVALIDIO       equ   00004H
PICKFLAG        equ   00002H
VBLNKFLG        equ   00001H
;
; Subsystem Control Register
;
GPCTRL_NC       equ   00000H
GPCTRL_ENAB     equ   04000H
GPCTRL_RESET    equ   08000H
```

```
CHPTEST_NC      equ    00000H
CHPTEST_NORMAL  equ    01000H
CHPTEST_ENAB    equ    02000H
IGPIDLE         equ    00800H
IINVALIDIO      equ    00400H
IPICKFLAG       equ    00200H
IVBLNKFLG       equ    00100H
RGPIDLE         equ    00008H
RINVALIDIO      equ    00004H
RPICKFLAG       equ    00002H
RVBLNKFLG       equ    00001H
;
; Current X, Y & Dest X, Y Mask
;
COORD_MASK      equ    007FFH

;
; Advanced Function Control Register
;
CLKSEL          equ    00004H
DISABPASSTHRU   equ    00001H
;
; Graphics Processor Status Register
;
GPBUSY          equ    00200H
DATDRDY         equ    00100H
;
; Command Register
;
CMD_NOP         equ    00000H
CMD_LINE        equ    02000H
CMD_RECT        equ    04000H
CMD_RECTV1      equ    06000H
CMD_RECTV2      equ    08000H
CMD_LINEAF      equ    0A000H
CMD_BITBLT      equ    0C000H
CMD_OP_MSK      EQU    0F000H
BYTSEQ          equ    01000H
_16BIT          equ    00200H
PCDATA          equ    00100H
INC_Y           equ    00080H
YMAJAXIS        equ    00040H
INC_X           equ    00020H
DRAW            equ    00010H
LINETYPE        equ    00008H
LASTPIX         equ    00004H
PLANAR          equ    00002H
WRTDATA         equ    00001H
;
;    Short Stroke Vector Transfer Register
;    (The angular defs also apply to the Command Register)
```

```
;
VECDIR_000      equ    00000H
VECDIR_045      equ    00020H
VECDIR_090      equ    00040H
VECDIR_135      equ    00060H
VECDIR_180      equ    00080H
VECDIR_225      equ    000A0H
VECDIR_270      equ    000C0H
VECDIR_315      equ    000E0H
SSVDRAW         equ    00010H
;
; Background Mix Register
;
BSS_BKGDCOL     equ    00000H
BSS_FRGDCOL     equ    00020H
BSS_PCDATA      equ    00040H
BSS_BITBLT      equ    00060H
;
; Foreground Mix Register
;
FSS_BKGDCOL     equ    00000H
FSS_FRGDCOL     equ    00020H
FSS_PCDATA      equ    00040H
FSS_BITBLT      equ    00060H
;
;    The Mixes (Note that some are duplicated for ease
;    of use.
;
MIX_MASK                    equ         0001FH

MIX_NOT_DST                 equ         00000H
MIX_0                       equ         00001H
MIX_1                       equ         00002H

MIX_DST                     equ         00003H
MIX_LEAVE_ALONE             equ         00003H

MIX_NOT_SRC                 equ         00004H

MIX_SRC_XOR_DST             equ         00005H
MIX_XOR                     equ         00005H

MIX_NOT__SRC_XOR_DST        equ         00006H
MIX_XNOR                    equ         00006H

MIX_SRC                     equ         00007H
MIX_REPLACE                 equ         00007H
MIX_PAINT                   equ         00007H

MIX_NOT_SRC_OR_NOT_DST      equ         00008H
MIX_NAND                    equ         00008H
```

```
MIX_NOT_SRC_OR_DST              equ         00009H
MIX_SRC_OR_NOT_DST              equ         0000AH

MIX_SRC_OR_DST                  equ         0000BH
MIX_OR                          equ         0000BH

MIX_SRC_AND_DST                 equ         0000CH
MIX_AND                         equ         0000CH

MIX_SRC_AND_NOT_DST             equ         0000DH
MIX_NOT_SRC_AND_DST             equ         0000EH

MIX_NOT_SRC_AND_NOT_DST         equ         0000FH
MIX_NOR                         equ         0000FH

MIX_MIN                         equ         00010H
MIX_DST_MINUS_SRC               equ         00011H
MIX_SRC_MINUS_DST               equ         00012H
MIX_PLUS                        equ         00013H
MIX_MAX                         equ         00014H
MIX_HALF _DST_MINUS_SRC         equ         00015H
MIX_HALF__SRC_MINUS_DST         equ         00016H
MIX_AVERAGE                     equ         00017H
MIX_DST_MINUS_SRC_SAT           equ         00018H
MIX_SRC_MINUS_DST_SAT           equ         0001AH
MIX_PLUS_SAT                    equ         0001BH
MIX_HALF__DST_MINUS_SRC_SAT     equ         0001CH
MIX_HALF__SRC_MINUS_DST_SAT     equ         0001EH
MIX_AVERAGE_SAT                 equ         0001FH
;
; Memory Control Register
;
BUFSWP          equ     00010H
VRTCFG_2        equ     00000H
VRTCFG_4        equ     00004H
VRTCFG_6        equ     00008H
VRTCFG_8        equ     0000CH
HORCFG_4        equ     00000H
HORCFG_5        equ     00001H
HORCFG_8        equ     00002H
HORCFG_10       equ     00003H
;
; Pixel Control Register
;
MIXSEL_FRGDMIX  equ     00000H
MIXSEL_PATT     equ     00040H
MIXSEL_EXPPC    equ     00080H
MIXSEL_EXPBLT   equ     000C0H
COLCMPOP_F      equ     00000H
COLCMPOP_T      equ     00008H
```

```
COLCMPOP_GE     equ    00010H
COLCMPOP_LT     equ    00018H
COLCMPOP_NE     equ    00020H
COLCMPOP_EQ     equ    00028H
COLCMPOP_LE     equ    00030H
COLCMPOP_GT     equ    00038H
PLANEMODE       equ    00004H

;
; Macros:
;

;
; OUTPW
;
; Outputs a value to an 8514/A register. May trash AX, DX. Last value
; and register are kept in AX and DX, respectively.
;
OUTPW       macro      register, value
            IFNB <value>
               IFDIFI <value>,<ax>
                 mov    AX, value
              ENDIF
            ENDIF
            IFNB <register>
               IFDIFI <register>,<dx>
              mov      DX, register
              ENDIF
            ENDIF
            out        DX, AX
            endm
;
; WAITQ
;
; Waits until the specified number of entries are available in the
; command queue. AX and DX are destroyed. No error checking on the
; input parameter is performed, but it must be > 0 AND <= 8
;
WAITQ       MACRO      qEntries
            LOCAL      CheckQ

            mov         DX, GP_STAT        ; Get the queue status address.
CheckQ:
            in          AL, DX             ; Get the status
                                            ; Is the queue entry available?
            test        AL, 0100H SHR (qEntries)
            jnz         CheckQ             ; Yes if result is
Zero.
            ENDM
;
; CLRCMD
```

```
;
; Waits until the no more commands are active.
; AX and DX will be destroyed.

CLRCMD      MACRO
            LOCAL     WaitLoop

            mov       DX, GP_STAT       ; Make sure the board's not busy...
WaitLoop:   in        AX, DX            ; Read the status.
            test      AX, BUSY          ; Is the machine busy?
            jnz       WaitLoop          ; Yes, try again.
            ENDM
```

Listing 7-11: Assembly-language routine to BitBLT mono data from PC to 8514/A

```
            page      60,132
;
; LIST7-11.ASM - Assembly routine to perform monochrome data expansion onto
;                the 8514/A.
;                 Written for Small Model C interface.
;
;                 Assemble with "MASM LIST7-11;"
;
;
            DOSSEG
            .MODEL    SMALL
            .286                        ; 8514/A only runs on 286 or higher

            include   REG8514.INC       ; Also known as LIST8-10.INC

;
; The following equates are used for _BLTMonoToBd to get
; parameters off the stack.
;
SRCPTR      equ       [BP+4]
DSTX        equ       [BP+6]
DSTY        equ       [BP+8]
PWIDTH      equ       [BP+10]
HEIGHT      equ       [BP+12]
ONCOLOR     equ       [BP+14]
OFFCOLOR    equ       [BP+16]

            .DATA
```

```
            .CODE

            public      _BLTMonoToBd


;
; _BLTMonoToBd(FAR *srcPtr, WORD dstX, WORD dstY, WORD width, WORD height,
;                WORD onColor, offColor);
;
; Expands an array of monochrome image information into color information:
; All 0 bits set the background color, all 1 bits set the foreground color.
; "dest" is ignored, since the dstX & dstY values
; indicate the destination position. AX, BX, CX, and DX are destroyed,
; the left and right clip edges are reset to 0 and 1023 respectively,
; the foreground and background mixes are set to MIX_SRC, and the color
; compare operation is set to FALSE. Colors are left in an undefined state.
; The write mask and top and bottom clip edge settings are honored.
;
; No error checking on null height or width or other bad parameters is
; performed.
;
;
_BLTMonoToBd proc
            push        BP
            mov         BP, SP
            push        SI
            push        DI

            mov         SI, SRCPTR          ; Point to source data area

            CLRCMD                          ; Wait for the current operation to end
            WAITQ       8                   ; Wait for queue to be completely empty
            mov         AX, DSTX            ; Get dstX
            mov         BX, AX              ; Save AX for now
            and         AX, 3               ; Get nugget index.
            add         AL, 3
            mov         DI, AX              ; Adjust and store for later
            mov         AX, BX
            and         AX, 03FFH           ; Mask and set left clip edge
            or          AX, SCISSORS_L
            OUTPW       MULTIFUNC_CNTL
            and         AX, 03FFH
            add         AX, PWIDTH          ; Add width
            dec         AX                  ; Calculate right clip edge
            and         AX, 03FFH
            or          AX, SCISSORS_R
            OUTPW                           ; DX is already set, as is AX
            mov         AX, PWIDTH          ; Get width again
            add         AX, 7               ; Round up to nearest byte boundary
            mov         BX, AX
            shr         BX, 3               ; Get byte count
```

```
            mov         PWIDTH, BX          ; Save it
            and         AX, 0FFF8H          ; Round up to nearest multiple of 8.
            dec         AX                  ; Adjust for register
            OUTPW       MAJ_AXIS_PCNT       ; Set it
;
; Setting the mixes is very important. The foreground source select is used
; for all PC bits that are set to 1, and the foreground mix indicates that
; the destination pixel should be updated.
;
; The background mix setting for the source doesn't really matter, since
; the mix operation tells it that all 0 bits should not affect the dest.
;
; Also important is telling the 8514/A that it should use monochrome PC Data
; to determine the mix results.
;
            OUTPW       FRGD_MIX, <MIX_SRC or FSS_FRGDCOL>  ; Set foreground mix
            OUTPW       BKGD_MIX, <MIX_SRC or BSS_BKGDCOL>  ; Set background mix
            OUTPW       MULTIFUNC_CNTL, <PIX_CNTL or MIXSEL_EXPPC or COLCMPOP_F>
            OUTPW       FRGD_COLOR, ONCOLOR  ; Set foreground color
            mov         CX, HEIGHT          ; Get line count
BM_Loop_Outer:
            OUTPW       BKGD_COLOR, OFFCOLOR ; Set every time because
                                             ;  this operation trashes it.
            OUTPW       CUR_X, DSTX         ; Set the starting position.
            OUTPW       CUR_Y, DSTY
            inc         AX                  ; Next line
            mov         DSTY, AX            ; Save it
            mov         AX, CMD_LINE or PCDATA or LINETYPE or DRAW or PLANAR or WRTDATA
            OUTPW       CMD                 ; Draw a hor. vect. using PC as source
            push        CX
            mov         CX, PWIDTH          ; Get byte width count
            xor         AX, AX
            mov         BX, DI              ; Get the nugget shift value.
            mov         DX, PIX_TRANS       ; Load just once for this loop.
BM_Loop_Inner:
            xchg        CX, BX              ; Need to use CX, and info in BL.
            lodsb                           ; Load a byte
            mov         CH, AL              ; Save it for later
            ror         AX, CL              ; Shift upper nibble into place
            OUTPW       ,                   ; Send to 8514/A, DX is still loaded
            rol         AX, 4               ; Shift lower nibble into place
            OUTPW       ,                   ; Send to 8514/A, DX still loaded
            mov         AH, CH              ; Use as upper byte of AX
            xchg        CX, BX              ; Restore CX
            loop        BM_Loop_Inner
            mov         CL, BL              ; Have to send out one last byte to
            ror         AX, CL              ;  make sure all data went to board
            OUTPW       ,                   ;  because it might've been mis-aligned
            pop         CX
            loop        BM_Loop_Outer
;
```

```
; Time to reset all the things we changed.
;
            WAITQ       8
             OUTPW       BKGD_MIX, <MIX_SRC or BSS_BKGDCOL> ; Set background mix
             OUTPW       MULTIFUNC_CNTL, <PIX_CNTL or MIXSEL_FRGDMIX or COLCMPOP_F>
            OUTPW       , <SCISSORS_L>
             OUTPW       , <SCISSORS_R or 03FFH>

            pop         DI
            pop         SI
            pop         BP
            ret
_BLTMonoToBd  endp

            end
```

Listing 7-12: C routine to call the assembly-language routine in Listing 7-11

```
/*---- --------     -----      ----        --      - --- -     -  --

    DoSomething()

    Demonstrate the use of the BLTMonoToBd function from Listing 7-11
    (LIST7-11.ASM).

-------- --                    --       -                  -- -- -- -- - *
VOID   DoSomething()
{
    static UBYTE    markBitmap[14][2] =
                {
                 0xFF, 0xFF,    /* 1111111111111111 */
                 0xC0, 0x03,    /* 1100000000000011 */
                 0xA0, 0x05,    /* 1010000000000101 */
                 0x90, 0x09,    /* 1001000000001001 */
                 0x88, 0x11,    /* 1000100000010001 */
                 0x87, 0xE1,    /* 1000011111100001 */
                 0x84, 0x21,    /* 1000010000100001 */
                 0x84, 0x21,    /* 1000010000100001 */
                 0x87, 0xE1,    /* 1000011111100001 */
                 0x88, 0x11,    /* 1000100000010001 */
                 0x90, 0x09,    /* 1001000000001001 */
                 0xA0, 0x05,    /* 1010000000000101 */
                 0xC0, 0x03,    /* 1100000000000011 */
                 0xFF, 0xFF     /* 1111111111111111 */
                };

            WORD    i;
```

```
    for(i = 0; i < 15; i++)                    /* Test the blit code code */
        BLTMonoToBd(markBitmap, 400, 14 * i, 16 - i, 14, i, 15 - i);

    for(i = 0; i < 15; i++)                    /* Test the blit code code */
        BLTMonoToBd(markBitmap, 450, 14 * i, 16 - i, 14, i, 15 - i);

    for(i = 0; i < 15; i++)    /* Test the blit code code */
        BLTMonoToBd(markBitmap, 500 + i, 14 * i, 16 - i, 14, i, 15 - i);

    for(i = 0; i < 15; i++)    /* Test the blit code code */
        BLTMonoToBd(markBitmap, 550 + i, 100 + 14 * i, 16, 14, i, 15 - i);


    return;
}
```

Area Fills

One primitive we have touched upon only briefly in the register documentation is the area fill. By specifying an outline, you can direct the 8514/A to fill that outline. Unfortunately, it's not quite as simple as it seems. The area fill is implemented via the fast-fill-rectangle operation. You have to set the PLANEMODE bit in the Pixel Control register, set the Read Mask register to select the source plane for the area fill, specify a rectangular region for the area-fill operation, and start the fast-fill-rectangle command. The 8514/A then starts scanning the source plane for bits that are set to 1. Initially, the fill is off, but when the first 1 bit is found, the fill state toggles on and continues to fill the destination region until the next edge (or 1 bit) is detected, at which time the state toggles off. This process repeats for each line in the specified rectangular region.

This is a quick implementation of the standard odd-/even-fill approach documented in tomes such as Foley and Van Dam's *Principles of Interactive Computer Graphics* (Addison-Wesley). There are some drawbacks, most apparent of which is that if you want to fill a polygon, you may have problems with vertices. In most cases, a vertex is only one pixel thick, which means that when the scan reaches that position, the fill mode toggles on but not immediately back off because there isn't a second pixel to trigger it. This problem can be solved, al-

beit inelegantly, by drawing the borders in XOR mode, so that vertices are turned off. Once the fill has finished, you then need to redraw the border with XOR mode disabled. Also, you must use the special outline drawing mode when defining the borders so that only one pixel per scan line gets drawn in memory. This way, you avoid the "stepping" problem inherent in "pixelated" line drawing — a situation in which multiple pixels get drawn on the same scan line if the line is relatively horizontal.

One other disadvantage of this approach for filling objects is that it requires an additional plane of memory be dedicated to its use. IBM, in the AI, gets around this problem by using off-screen memory to define the extra plane. Because there isn't enough pixel memory for the extra plane there, though, IBM resorted to dividing the display into thirds (256 lines apiece in 1024 x 768 mode) and then drawing the definitions into that space, each into its own plane. The IBM AI performs three separate area fills — one for each of the three off-screen planes — and then does a single-plane expansion BLT to the display from each of those areas. This approach can get rather messy, plus it means that you still have to sacrifice a good portion of off-screen memory.

Summary

Although the 8514/A is somewhat strange in its implementation, once you understand how it operates, you can make it do some interesting things. This chapter is intended to provide you with an overview of how to get started using the 8514/A registers in your graphics programs.

If you would like additional information, we recommend you contact the companies listed in Chapter 8 and Appendix A for copies of their register specifications.

8

8514/A Hardware Compatibles

This chapter provides a brief overview of the additional capabilities, beyond 100% compatibility with IBM's 8514/A, offered by several companies in their 8514/A-compatible chip sets. Each of the four companies from which we have received detailed technical specifications has its own section below. At the time of this writing, none of the companies mentioned had been able to supply us with working hardware. These summaries are based solely on the specifications they provided to us, and some features may change in the finished product. Therefore, caveat emptor.

We have refrained from documenting specific register-based differences, partly because they may change and partly because some of the companies requested that we not do so. Not all the companies are currently planning register-accessible enhancements.

The four companies whose products we summarize are ATI Technologies, Chips and Technologies, Headland Technology, and Western Digital Imaging.

In addition, Integrated Information Technologies (IIT), Trident, and SIS Microelectronics have announced 8514/A-compatible chip sets, but they were not able to supply specifications in time for us to include details here. We have managed to provide some information about these products in this chapter.

But First . . . Compatibility and Extensions

Before we discuss the clone enhancements, we want to point out that *compatibility* is a broad term. It can mean many things, and extensions may not always be the blessings they initially appear to be.

Case in Point: The VGA

When the VGA was first cloned, several companies hit the market with VGA BIOS compatibles that were actually EGAs with BIOS patches to emulate a VGA. What this really meant was that the BIOS commands were compatible but the hardware wasn't.

Even when the first VGA chip sets started appearing, compatibility problems arose. Some of these problems were insignificant, such as the lack of the testing register whose sole purpose was to toggle the state of an LED on the board. This "feature" was used only in testing and had no practical application in the real world. Some compatibility problems were more severe, such as VGA-compatible boards that wouldn't support Microsoft Windows or OS/2 because of flaws in their initialization logic. It took well over a year to iron out these problems, and customers of the original boards were in many cases out of luck.

Next were the extensions to the IBM VGA standard — namely, higher resolutions. Resolutions for 16-color modes have reached 1024 x 768, and in many cases, resolutions of 800 x 600 are commonly achieved in 256-color modes. Some companies have added special addressing modes, and others have added features such as hardware cursors. These extensions are generally useful, but there is one overriding problem with them — there is no standard for the extensions. There are, by our count, at least a dozen VGA-chip manufacturers, each with its own method of selecting resolutions and switching memory banks. Also, many, if not most, VGA-board companies modify the chip-company-supplied VGA BIOS to suit their own needs, with the first change usually being the mode numbers to initialize the extended res-

olutions. Combine these two "enhancements" with the need to support all the popular VGAs in your application software, and you have a ridiculous and virtually insurmountable programming problem.

VGA companies use these nonstandard "enhancements" as a way to differentiate themselves from their competitors. And if these companies could provide all necessary application drivers for their boards, this differentiation would be acceptable. But because these companies generally cannot support every single package users want to use, the differentiation ends up hurting users, not to mention what it does to software developers.

Fortunately, there appears to be a little bit of light at the end of the tunnel. The Video Electronics Standards Association (VESA) has recently passed two standards that will make it easier for VGA software developers to support the myriad of VGA boards available — providing that VGA-board companies adhere to those standards.

The 8514/A

Much of the history of VGA is repeating itself with the 8514/A. First, many companies began to offer 8514/A AI compatibles. These boards were all based on the TI 34010. Lately, some companies have even been looking into supporting the AI on VGA boards.

Next, the 8514/A-clone companies have introduced what should be 8514/A-compatible chips with custom enhancements. As we pointed out earlier, the 8514/A hardware from these companies and their OEMs is not yet readily available, and therefore it will be a while before the chip sets can be put to the test. With the complexity and undocumented capabilities of the 8514/A, it will be quite some time before we can be sure that these clone chips are 100% IBM 8514/A compatible.

The enhancements seem to be following the same path as the VGA enhancements. In some cases, the enhancements are transparent, which

is of benefit to end users and developers alike. It's a different story for the nontransparent enhancements, which are in the majority. Although ATI, Headland, and Western Digital have all implemented quick-setup line drawing, none of the three companies' chip sets are compatible in their support of that function, which means that in general only the drivers shipped by the companies themselves will be able to take advantage of it. To top it off, there is currently no known test a program can run to tell which board it's running on.

It is still early enough in the game, however, for changes to be made to the second-generation silicon, and VESA's 8514/A standards subcommittee will hopefully play an important part in standardizing on an enhanced 8514/A. Without a standard set of extensions, enhanced 8514/As are bound to have no real outside support other than from system integrators.

Transparent vs. Nontransparent Enhancements

In the following sections, we describe the enhancements that each company has told us its 8514/A-compatible chip set will have. There are two types of enhancements: transparent ones, which speed up any program that runs on the board without changes to the program; and nontransparent ones, which require board-specific code to be usable. And because there's currently no documented way for a program to distinguish what board it's running on, even developers willing to go to the work of putting board-specific code in their drivers have no way of telling which subroutines to run.

Most of the transparent enhancements are changes in the hardware, and either improve the speed of a board or add configuration flexibility to it. Changes include the ability to support noninterlaced monitors at 1024 x 768; support for DRAM instead of, or in addition to, VRAM; and VGA support on the same chip or board as the 8514/A. These changes do not generally affect how software runs on the board.

Nontransparent enhancements are accessed through specific software. These include quick line setups (described above), with incompatible improvements on each board; readable registers; and new registers not available on IBM's board. These features are currently accessible only to drivers written by the board manufacturers or that are otherwise targeted only for use with that board. There is some hope that the VESA 8514/A subcommittee will standardize some of these features in time for a wave of new products with compatible extensions to reach the market, but every board built and sold up to that time is a board that will be incompatible with any new extended standard. For now, drivers from outside the board companies themselves are expected to be least-common-denominator boards written to the AI or the IBM hardware specification.

When writing a product to run on all 8514/A-compatible boards, it's important to test the product thoroughly on the original 8514/A. This is partly to make sure the program will run on all compatibles and partly to ensure the product runs acceptably fast on the IBM board. If it doesn't run fast enough, it won't be usable by a large segment of the market: IBM-board owners and those who own the slowest of the DRAM-based compatible boards.

The following sections describe the products that are available or about to reach the market at this writing. We hope to offer revised editions of this book with more detailed technical information about these products and additional ones as it becomes available.

ATI Technologies

ATI, located in Scarborough, Ontario, Canada, was one of the quiet entries into the 8514/A-clone race. Months after many other companies had loudly announced their product intentions, ATI leaked word of its Mach 8 8514/A-compatible chip set and, a few months later, announced products that incorporated the Mach 8.

The Mach 8 Graphics ASIC Chip appears to be the most full featured of the ones we have examined. Its list of enhancements is impressive:

• Quick-Setup Line Drawing. This eliminates the need to calculate Bresenham line parameters. Instead, you do what seems natural — namely, specify the start and end points of the line — and the chip does the rest. This is critical for achieving high performance when drawing very short lines, as you might in a CAD application.

• Preclipping. When the clipping rectangle is set in the 8514/A, each pixel to be drawn is compared against it, and if the pixel falls outside the clipping rectangle, it is discarded. This technique is commonly called postclipping. The software must still ensure that line end points are in the range –512 to 1535. The Mach 8 preclipping hardware draws only those lines that pass inside the clipping rectangle. Lines that fall completely outside the rectangle are discarded without further processing. Although the 8514/A requires coordinates to fall in the range –512 to 1535, the Mach 8 accepts coordinates in the range –32,768 to 32,767. This significantly improves line-drawing performance. The preclipping avoids the overhead in processing non-drawing portions of a line.

• Packed Monochrome Data Transfers. As you may have noticed in Chapter 7, transferring monochrome information to and from the 8514/A is a painful job, primarily because information has to be sent four bits at a time from an awkward bit position. The Mach 8 chip offers the ability to pack monochrome information such that 8 bits of monochrome data can be transferred simultaneously (16 bits in word mode).

• Gray-Scale Font Support. ATI provides a method for generating anti-aliased scalable fonts, which it calls the ATI Crystal Font Technology, but no documentation is available at this time, apparently because of the technology's proprietary nature. Initial demos of the product show good performance and very high text quality.

• Source Color-Comparison Registers. These work in a similar way to regular color-comparisons, but they are applied to the source color value, not the destination color value.

• DRAM and VRAM Support. The Mach 8 supports both DRAM and VRAM configurations. Thus ATI can use one chip to produce low- and high-cost (and performance) products.

• Variable-Length Monochrome and Color Patterns. These let you use patterns of monochrome data, instead of the normal source, for expansion, drawing, or color patterns.

• Bounding-Box Determination. The Mach 8 provides a facility that lets you determine the maximum and minimum (x, y) coordinates of a polyline, just by stuffing the information into the board. This is generally much faster than setting up a loop to compare and update on the PC.

• Improved BitBLTs. When using the Mach 8's special BLT registers, you don't have to worry about BLT direction because the chip takes care of overlap automatically.

• Flash Fill. This takes advantage of a VRAM chip's ability to have its contents set to a specific value. ATI claims that up to 4K of data can be filled every 200 nanoseconds. This is normally used for clearing the display and will not work on DRAM boards.

• Shadow CRT-Control Registers. The Mach 8 allows nonstandard monitors to be attached to the Mach 8-based board. Register-based software will think it is setting up the board for a standard IBM monitor type, but the actual values used will have already been set by the configuration software and locked into place. This allows noninterlaced monitor operation to occur transparently with software written specifically for IBM's 8514/A, which does not really support noninterlaced 1024 x 768 display.

• Increased Command-Queue Size. ATI has doubled the size of the command queue from 8 slots to 16. This means the PC can do more work while the graphics processor is busy with a full queue, thereby improving system performance.

Detailed information about the Mach 8 can be obtained from ATI. (See Appendix A for address information.)

Chips and Technologies

Chips and Technologies is the granddaddy of cloning in the PC marketplace. Initially producing EGA and motherboard chip sets, it is now one of the companies to enter the 8514/A-clone arena.

The company's philosophy generally seems to be that the first-generation clone should be 100% compatible with the hardware being cloned, with no real additional frills, unless the frills can be introduced transparently. Its 82C480 chip is no exception.

As such, many of the enhancements Chips and Technologies has provided are not immediately apparent, but almost all are readily available for programs to use without the software knowing how to take advantage of them, which is the benefit of adding transparent enhancements.

One of the remarkable things that Chips and Technologies has revealed has to do with performance. In its research, it determined that when application-level benchmarks were performed using AI-based applications, the 8514/A graphics processor was never active for more than 6% of the time. This knowledge allowed it to focus on a more economical solution without having to worry much about performance issues. Even so, the performance of the 82C480 is said to exceed that of the 8514/A by as much as two times.

The enhancements Chips and Technologies has introduced are as follows:

• Separation of High-Speed Chip Functions. Such things as clock selects and video shift registers have been left out of the 82C480 so that it could be produced inexpensively. This also has the advantage of removing the display-frequency limitations, allowing video frequencies of up to 300 MHz and resolutions of up to 2560 x 2048.

• Greatly Decreased Chip Count. By combining all the support glue logic required by the 82C480 into a single chip (the 82B484), Chips and Technologies can build a 512K 8514/A-compatible board with only nine chips, which reduces board costs substantially. These nine chips include four VRAMs, the 82C480, the 82B484, and the RAMDAC.

• Deeper Queue. Chips and Technologies has internally enlarged the queue from 8 words to 64 words. The net effect of this is that the Graphics Processor Status register indicates the queue is almost always empty. Programs therefore spend less time polling the queue status and can get on with other things.

• Shadow CRT Registers. Chips and Technologies has a feature similar to ATI's, in that a second set of CRT-control registers exists. This second set is in fact the master CRT-control register set and can be updated only with specially written software. This feature lets existing 8514/A software run on noninterlaced displays.

• Readable Registers. The 82C480 supports the reading of all registers. Most of the 8514/A's registers are write only.

Chips and Technologies provided us with a statement that explains why it is pursuing only transparent enhancements at this time. It feels that the headaches caused by incompatible VGA extensions could occur in the 8514/A-clone market as well. The net result is that the 82C480 architecture allows for extensions, but only transparent extensions or future extensions approved by the VESA 8514/A subcommittee

will be added. We applaud Chips and Technologies' decision to help propagate only truly standard extensions.

More information on the 82C480 and 82B484 can be obtained by writing to or calling Chips and Technologies. (See Appendix A for address information.)

Headland Technology

Headland Technology is the product of the merger of G2, a motherboard chip company, and Video Seven, one of the EGA and VGA market leaders. Video Seven now exists only as a brand name for graphics boards.

Headland was the first company to publicize the fact that it is possible to reverse-engineer the 8514/A, and its specification was the first one available, though only under a nondisclosure agreement. Ironically, Headland is also one of the last major graphics companies to come to market with an 8514/A-clone chip set, the HT250. The HT250 consists of three chips — a master, a pixel processor, and a display chip. The PC-bus interface, CRT controller, and graphics processor are implemented in the master chip; the pixel processor performs data manipulation; and the display chip handles the transfer of information to the display as well as some new features.

Headland's enhancements are equal to ATI's in diversity, and whereas ATI seems to have focused more on font-related enhancements, Headland has applied itself more to graphical environments, Graphical User Interface (GUI), and high-color-content graphics. The enhancements it provides are:

• Quick-Setup Line Drawing. Headland's implementation is similar to ATI's, in the sense that you can easily specify start and ending points

for a line and then draw it without the messy Bresenham parameter calculations.

• Direct Memory Access. This feature lets you address 8514/A display memory locally, without having to deal with the inadequate Pixel Data Transfer register. This is extremely useful for uploading and downloading image data.

• 32-Bit Register Access. To allow 386 and 486 systems to use 32-bit data transfers, the HT250 chip set supports 32-bit access, which is useful when combined with the following feature.

• Variable Pixel Depth. By using additional pixel-processor chips on a board, pixel depths up to 32 bits per pixel can be achieved. This means that true-color graphics boards with incredibly fast drawing capabilities are just around the corner, because the 8514/A's psuedo-planar architecture allows 32-bit pixels to be updated just as quickly as 8-bit pixels.

• Display Configuration Protection. Just like ATI and Chips and Technologies, Headland provides a mechanism by which a secondary set of CRT-control registers can override the ones a regular 8514/A program might use, enabling support for noninterlaced 1024 x 768 displays.

• Fixed 640 x 480 256-Color Mode. Using the HT250 extensions, you no longer have to translate the *y* coordinate in the 4-bit-per-pixel 640 x 480 mode.

• Hardware Cursor. Headland has added the ability to define and manipulate a nondestructive hardware cursor up to 64 pixels square. The cursor colors and masks can be defined independently of one another.

• Readable Registers. All 8514/A and extended registers are readable.

• Increased Queue Size. Headland has added six more words to the command queue, for a total of 14 words.

• Area Pattern. You can define an area pattern for use in most drawing operations.

• Packed Bit Transfers. Monochrome bit-map information of 8 or 16 bits can be sent to the board with each write, as opposed to the 8514/A's 4 or 8 bits. This increases the bandwidth of monochrome-to-color expansions.

• Data Compression/Decompression. The HT250 supports built-in run-length encoding and decoding, thereby increasing the speed with which images can be transferred while decreasing the byte size of the images.

• Context Switching. Up to four independent sets of registers can be manipulated by a program or a multitasking engine. This applies primarily to the scissor registers, pixel-processing registers, and coordinate registers. Additionally, the four sets of scissors registers can be activated to be operational at the same time, thereby allowing for complex clipping areas.

• Delta Vectors. For programs that rely upon relative coordinates instead of absolute coordinates for line drawing, differences of –127 to +128 can be added to each coordinate for each new line.

More information on the HT250 chip set can be obtained by writing to or calling Headland Technology. (See Appendix A for address information.)

Western Digital Imaging (WDI)

WDI is the result of Western Digital's acquisitions of Verticom, a high-end graphics-board company, and Paradise, the current market leader in VGA-chip sales.

As with Headland and Video Seven, the Paradise name is now used only for retail products, and OEMs end up buying WDI chips. In the case of 8514/A-compatible chip sets, WDI has named its two-chip solution the PWGA-1.

WDI's entry seems to add the least new features to the chip set, but that impression might just be due to the documentation we have received. On the other hand, WDI was the first company to ship a finished product, as well as the first company to publicly release register information.

WDI's added features and services for the PWGA-1 include:

• Shadow CRT Registers. This is the one enhancement that has appeared consistently in all the chip sets. It allows program-based CRT register programming to be overridden transparently. WDI provides two additional sets — one for 640 x 480 resolution and one for 1024 x 768 resolution.

• Quick-Setup Line Drawing. Again, this feature reduces the overhead of line-drawing operations . All that's needed are the starting and ending points of the line.

• Patterned Lines. WDI has added real line patterns to the PWGA-1, which speeds up that operation considerably. On a normal 8514/A, the PC has to generate line patterns a pixel at a time.

• Turbo BitBLTs. In the 4-bit-per-pixel mode, WDI has taken advantage of the extra bandwidth available and has doubled the speed of

BitBLTs. This has limited application, as it works only in 16-color mode.

• Double-Buffered Display. In 1024 x 768 mode, two pages of 1024 x 768 resolution can be made available to assist in double-buffering, a technique used for animation and backups of the display for "undo" in paint programs.

• Software Tool Kit. WDI is the only company so far to offer a comprehensive software tool kit. This tool kit offers libraries of quick routines to perform all the functions the board is capable of. This makes it a lot easier to program the PWGA-1, but the negative side is that it is similar in concept to the AI, albeit somewhat faster.

More information on the PWGA-1 chip set can be obtained by writing to or calling Western Digital Imaging. (See Appendix A for address information.)

Integrated Information Technologies (IIT) and Trident

These two companies are sharing VGA and 8514/A technology for their combined product and therefore offer the same 8514/A-compatible chip. The special features reportedly offered by this one-chip 8514/A include integrated VGA and DRAM support for lower cost. IIT claims that at the chip's core is a 25-MIPS RISC engine that allows for performance comparable to IBM's product by using DRAMs instead of the more costly VRAMs. Building a VGA into the chip is another way to reduce board cost, because the use of other 8514/A boards requires some type of IBM standard character-display system, usually a VGA.

See Appendix A for address information.

SIS Microelectronics

SIS has taken a unique approach to the 8514/A-compatible chip-set market by allowing for customized enhancements. This means that it is targeting only OEM (original-equipment manufacturer) companies and not end users. The client company gets to pick and choose which features it wants in the chip set, and SIS uses its ASIC libraries to put them together in a one- or two-chip package. SIS does the design and VLSI Technology provides the chip-manufacturing and shipping facilities for these custom chips.

See Appendix A for address information.

Appendix A

Company Addresses

ATI Technologies Inc.
3761 Victoria Park Ave.
Scarborough, Ontario
Canada M1W 3S2
(416) 756-0718

Chips and Technologies
3050 Zanker Rd.
San Jose, CA
95134
(408) 434-0600

Headland Technology, Inc.
46335 Landing Parkway
Fremont, CA
94538
(415) 656-7800

Integrated Information
Technology
2540 Mission College Blvd.
Santa Clara, CA
95054
(408) 727-1885

Texas Instruments
13500 N. Central Expressway
Dallas, TX
75265
(800) 232-3200

Trident Microsystems, Inc.
321 Soquel Way
Sunnyvale, CA
94086
(408) 738-3194

Video Electronics Standards
Association (VESA)
1330 South Bascom Avenue,
Suite D
San Jose, CA 95128-4502

Western Digital Imaging
800 E. Middlefield Rd.
Mountain View, CA
94043
(415) 474-2033

Appendix B

AI Commands by Function Number

Function Name	Function Number	Function Name	Function Number
HLINE	0	HQDPS	31
HCLINE	1	HRECT	32
HRLINE	2	HSBP	33
HCRLINE	3	HCLOSE	34
HSCP	4	HESC	35
HBAR	5	HXLATE	36
HEAR	6	HSCS	37
HSCOL	7	HCHST	38
HOPEN	8	HCCHST	39
HSMX	9	ABLKMFI	40
HSBCOL	10	ABLKCGA	41
HSLT	11	AERASE	42
HSLW	12	ASCROLL	43
HEGS	13	ACURSOR	44
HSGQ	14	ASCUR	45
HSCMP	15	ASFONT	46
HINT	16	AXLATE	47
HSPATTO	17	HINIT	48
HSPATT	18	HSYNC	49
HLDPAL	19	HMRK	50
HSHS	20	HCMRK	51
HBBW	21	HSMARK	52
HCBBW	22	HSLPC	53
HBBR	23	HRLPC	54
HBBCHN	24	HQCP	55
HBBC	25	HQDFPAL	56
HSCOORD	26	HSPAL	57
HQCOORD	27	HRPAL	58
HSMODE	28	ASCELL	60
HQMODE	29		
HQMODES	30		

Appendix C

AI-Compatible Character Definitions

The Adapter Interface supports three different types of fonts:

1) Short Stroke Vector Fonts.

Short stroke vector fonts (or SSV fonts) are made up of a number of lines or vectors that are no more than 15 pixels in length. Each SSV must be drawn at an angle that is a multiple of 45 degrees. Therefore, only vertical, horizontal, and diagonal lines can be drawn. For each SSV in a font character definition, you can indicate that it should draw or not draw. The latter case is handy for moving the drawing pointer without having to actually draw pixels.

Coincidentally, the 8514/A has the ability to take in short stroke vector information directly via the Short Stroke Vector Register.

The benefit of the SSV font type is that it is very quick to draw, and is reasonably easy to scale in size, since it is a vector definition and not a raster definition. Also, SSV fonts can have odd cell dimension. On the other hand, SSV fonts can be rather space consuming, and require that you clear the cell the character is to be drawn in, since otherwise you will end up with overwritten characters that will, in all likelihood, not be legible.

The only type of fonts that IBM supplies with their 8514/A are SSV fonts.

2) Monochrome bitmapped fonts.

Monochrome bitmapped fonts have been around for a long time. In the PC environment, the fonts used in the graphics modes on all industry standard graphics boards (CGA, EGA, MCGA, and VGA). This is basically a predefined block of bits, with set bits indicating pixels that are to drawn in the foreground color, and cleared bits indicating pixel that should be drawn in the background color.

The cell width of monochrome bitmapped fonts must be a multiple of 4, and the height a multiple of 2.

3) Color bitmapped fonts.

Color bitmapped fonts consist of three monochrome bitmaps, and therefore each pixel in a color font can have one of 8 colors. The actual colors are determined by the HXLATE call in the AI.

When using the AI text functions to display color bitmapped fonts, there are two ways in which the text can be displayed, both of which depend on the foreground color the text is to be drawn in.

If the foreground color is 0xFFFFFFFF (all bits set), the three monochrome bitmaps will be OR'd together, and the result will be treated like the monochrome bitmapped font usually is. If the foreground color is anything but 0xFFFFFFFF, then the three planes will determine which of the 8 colors defined by the HXLATE function a given pixel will be drawn in.

This font type cannot be used with the AI's alphanumeric functions.

All the fonts share the same header structure, but with variations in the contents. You pass a pointer to the header to functions like HSCS and ASFONT. The header is usually part of a larger font file that

contains several variations of a given font for different languages. The header of the font file contains information that points to the proper character set header. The file header looks like this:

Offset	Type	Description
0	WORD	Number of font variations (code pages) in file
2	WORD	The default code page number
4	WORD	Alternate default code page number

The following is repeated once for every code page:

6	BYTE	Code page ID, contains IBM specific code page info
10	WORD	Offset from start of file to the character set header for this page.

Once the proper code page has been located, and the address of the appropriate character set header has been determined, use the following structure. Note that the offset value is only a reference from the start of the header, not the file:

Offset	Type	Description
0	BYTE	Reserved
1	BYTE	Type of font: 0 = Bitmapped (monochrome and/or color) 3 = Short stroke vector All others are reserved
2	BYTE	Reserved
3	DWORD	Reserved
7	BYTE	Character width
8	BYTE	Character height
9	BYTE	Reserved
10	WORD	Character area in bytes Calculated by: ((width + 7) / 8) * height Only applies to bitmapped fonts
12	WORD	Flags Bit 15 - Reserved Bit 14 0 = Monochrome bitmapped font 1 = Color bitmapped font Bit 13 0 = Fixed spaced 1 = Proportional spaced Bits 12-0 - Reserved
14	DWORD	Offset of index table
18	DWORD	Offset of prop. spacing table
22	BYTE	ASCII value of first character in font
23	BYTE	ASCII value of last character in font
24	DWORD	Offset of character definition 1
28	WORD	Reserved
30	DWORD	Offset of character definition 2
34	WORD	Reserved
36	DWORD	Offset of character definition 3

All offsets are unsigned absolute byte offsets from the start of the file.

Character definitions 2 and 3 are only used by color bitmapped fonts, and contain the second and third planes of the font respectively.

The address of the actual bitmap or SSV definition for a given font character is calculated by using the ASCII character value as an index (after normalizing with the first character in the font) into the

index table. The contents of the location in the index table are then added to the offset of the appropriate character definition, resulting in the address of the font definition for the character in question.

Each character definition starts on a byte boundary. For SSV fonts, each SSV request occupies one byte, and terminates with a byte whose value is 0. Bitmapped fonts have the width rounded up to the nearest byte count (byte count = (width + 7) / 8) for each line in the definition.

The proportional spacing table is used to properly space variable width fonts. There are two bytes for each proportionally spaced character in a font, the first byte contains the left adjustment, and the second byte contains the right adjustment. The left adjustment should be subtracted from the X position at which the character is being drawn, and the right adjustment should be subtracted from the font width to determine the position to place the next character.

Appendix D

IBM AFI.H


```
/*
   IBMAFI.H

   Version of IBM's ibmafi.h rewritten for use with the book
   "Programming the 8514/A" by Jake Richter and Bud Smith

   (C) Copyright 1989, 1990 Panacea Inc.  All Rights Reserved

   This code may be freely incorporated in any program without
   royalty, as long as the copyright notice stays intact.

   This file contains definitions used by programs writing to the
   8514/A Adapter Interface.
*/

/*--------------------------------------------------------------------------

   Standard C Definitions

-------------------------------------- ---------------------------------*/
#define  FALSE          0
#define  TRUE           !(FALSE)
#ifndef NULL
#define  NULL           (void *)0
#endif

#define  BYTE           char
#define  WORD           short
#define  BOOL           short
#define  DWORD          long
#define  UBYTE          unsigned BYTE
#define  UWORD          unsigned WORD
#define  UDWORD         unsigned DWORD
#define  FAR            far
#define  HUGE           huge
#define  VOID           void

#define  MIN(a,b)       ((a) > (b) ? (b) : (a))
#define  MAX(a,b)       ((a) < (b) ? (b) : (a))
```

```
/*--------------- ------------ -- ---------------------------------------

   Standard AI Definitions

 ------------ ---- ------------------------------------------------------*/
typedef struct
   {
   WORD     x_coord;
   WORD     y_coord;
   } coord_pr;

typedef struct
   {
   BYTE     rx_coord;
   BYTE     ry_coord;
   } rcoord_pr;

struct afi_entries
{
  VOID(FAR *array[61])();
};


/*--------------------- ---- ------------------------- ------------------

   Externals - defined in CALLAFI.ASM

----------------- ---- --------------- ------------------ ----- -----*/
extern struct afi_entries FAR *afi_ptr;
extern FAR *(FAR getafi(VOID));
extern VOID FAR callafi(WORD, UBYTE FAR *);


/*---------    ------    - --------- --  - ------------------------ ---   ---

   Macro Function Definitions - sorted by command number.

--- - ----- ----------------- -- ------ ------ ------ -    -- -------- -*/
#define HLINE(X)     callafi( 0, (UBYTE FAR *) X)
#define HCLINE(X)    callafi( 1, (UBYTE FAR *) X)
#define HRLINE(X)    callafi( 2, (UBYTE FAR *) X)
#define HCRLINE(X)   callafi( 3, (UBYTE FAR *) X)
#define HSCP(X)      callafi( 4, (UBYTE FAR *) X)
#define HBAR()       callafi( 5, (UBYTE FAR *) 0L)
#define HEAR(X)      callafi( 6, (UBYTE FAR *) X)
#define HSCOL(X)     callafi( 7, (UBYTE FAR *) X)
#define HOPEN(X)     callafi( 8, (UBYTE FAR *) X)
#define HSMX(X)      callafi( 9, (UBYTE FAR *) X)
#define HSBCOL(X)    callafi(10, (UBYTE FAR *) X)
#define HSLT(X)      callafi(11, (UBYTE FAR *) X)
#define HSLW(X)      callafi(12, (UBYTE FAR *) X)
```

```
#define HEGS()          callafi(13, (UBYTE FAR *) 0L)
#define HSGQ(X)         callafi(14, (UBYTE FAR *) X)
#define HSCMP(X)        callafi(15, (UBYTE FAR *) X)
#define HINT(X)         callafi(16, (UBYTE FAR *) X)
#define HSPATTO(X)      callafi(17, (UBYTE FAR *) X)
#define HSPATT(X)       callafi(18, (UBYTE FAR *) X)
#define HLDPAL(X)       callafi(19, (UBYTE FAR *) X)
#define HSHS(X)         callafi(20, (UBYTE FAR *) X)
#define HBBW(X)         callafi(21, (UBYTE FAR *) X)
#define HCBBW(X)        callafi(22, (UBYTE FAR *) X)
#define HBBR(X)         callafi(23, (UBYTE FAR *) X)
#define HBBCHN(X)       callafi(24, (UBYTE FAR *) X)
#define HBBC(X)         callafi(25, (UBYTE FAR *) X)
#define HSCOORD(X)      callafi(26, (UBYTE FAR *) X)
#define HQCOORD(X)      callafi(27, (UBYTE FAR *) X)
#define HSMODE(X)       callafi(28, (UBYTE FAR *) X)
#define HQMODE(X)       callafi(29, (UBYTE FAR *) X)
#define HQMODES(X)      callafi(30, (UBYTE FAR *) X)
#define HQDPS(X)        callafi(31, (UBYTE FAR *) X)
#define HRECT(X)        callafi(32, (UBYTE FAR *) X)
#define HSBP(X)         callafi(33, (UBYTE FAR *) X)
#define HCLOSE(X)       callafi(34, (UBYTE FAR *) X)
#define HESC()          callafi(35, (UBYTE FAR *) 0L)
#define HXLATE(X)       callafi(36, (UBYTE FAR *) X)
#define HSCS(X)         callafi(37, (UBYTE FAR *) X)
#define HCHST(X)        callafi(38, (UBYTE FAR *) X)
#define HCCHST(X)       callafi(39, (UBYTE FAR *) X)
#define ABLKMFI(X)      callafi(40, (UBYTE FAR *) X)
#define ABLKCGA(X)      callafi(41, (UBYTE FAR *) X)
#define AERASE(X)       callafi(42, (UBYTE FAR *) X)
#define ASCROLL(X)      callafi(43, (UBYTE FAR *) X)
#define ACURSOR(X)      callafi(44, (UBYTE FAR *) X)
#define ASCUR(X)        callafi(45, (UBYTE FAR *) X)
#define ASFONT(X)       callafi(46, (UBYTE FAR *) X)
#define AXLATE(X)       callafi(47, (UBYTE FAR *) X)
#define HINIT(X)        callafi(48, (UBYTE FAR *) X)
#define HSYNC(X)        callafi(49, (UBYTE FAR *) X)
#define HMRK(X)         callafi(50, (UBYTE FAR *) X)
#define HCMRK(X)        callafi(51, (UBYTE FAR *) X)
#define HSMARK(X)       callafi(52, (UBYTE FAR *) X)
#define HSLPC()         callafi(53, (UBYTE FAR *) 0L)
#define HRLPC()         callafi(54, (UBYTE FAR *) 0L)
#define HQCP(X)         callafi(55, (UBYTE FAR *) X)
#define HQDFPAL(X)      callafi(56, (UBYTE FAR *) X)
#define HSPAL(X)        callafi(57, (UBYTE FAR *) X)
#define HRPAL(X)        callafi(58, (UBYTE FAR *) X)
   /* There is no function number 59 */
#define ASCELL(X)       callafi(60, (UBYTE FAR *) X)
```

```
/*-------------------------------------------------------------------------

   Other include files

-------------------------------------------------------------------------*/
#include "afidata.h"

/*-------------------------------------------------------------------------

   End of IBMAFI.H

-------------------------------------------------------------------------*/
```

Appendix E

AFI Data


```
/*
    AFIDATA.H

    Version of IBM's afidata.h rewritten for use with the book
    "Programming the 8514/A" by Jake Richter and Bud Smith

    (C) Copyright 1989, 1990 Panacea Inc.   All Rights Reserved

    This code may be freely incorporated in any program without
    royalty, as long as the copyright notice stays intact.


    This file contains structures used by programs writing to the
    8514/A Adapter Interface. This file is included by IBMAFI.H

    Note: In some cases it is necessary to have definitions that differ
          between the 4 and 8 bit per pixel versions of the AI. In such
          cases, the definitions for the 4 bit per pixel mode start with
          "B4_", and the 8 bit per pixels ones start with "B8_".
*/

/*-------------------------------------------------------------------------

    AI Constant Definitions

-------------------------------------------------------------------------*/
/*
    Color values for 4 and 8 bits per pixel using standard palette.
*/
#define  B4_RED        0x000CL
#define  B4_GREEN      0x000AL
#define  B4_BLUE       0x0009L
#define  B4_WHITE      (B4_RED | B4_GREEN | B4_BLUE)
#define  B4_CYAN       (B4_GREEN | B4_BLUE)
#define  B4_YELLO      (B4_RED | B4_GREEN)
#define  B4_MAGEN      (B4_RED | B4_BLUE)

#define  B8_RED        0x0049L
#define  B8_GREEN      0x0092L
```

```
#define  B8_BLUE      0x0024L
#define  B8_WHITE     (B8_RED | B8_GREEN | B8_BLUE)
#define  B8_CYAN      (B8_GREEN | B8_BLUE)
#define  B8_YELLO     (B8_RED | B8_GREEN)
#define  B8_MAGEN     (B8_RED | B8_BLUE)

/*
    Bit level definitions for each color (and intensity in case of 4 bpp).
    Used to control the individual bits that control a given color.
    When in 4 bit per pixel (bpp) mode, the colors are based on an IRGB
    palette similar to that of the CGA color text mode.
    When in 8bpp mode, 3 bits each are allocated to the Red and Green
    components, and 2 bits to the Blue.
*/
#define  B4_R0        0x004L
#define  B4_B0        0x001L
#define  B4_G0        0x002L
#define  B4_IN        0x008L

#define  B8_R0        0x001L
#define  B8_R1        0x008L
#define  B8_R2        0x040L
#define  B8_B0        0x004L
#define  B8_B1        0x020L
#define  B8_G0        0x002L
#define  B8_G1        0x010L
#define  B8_G2        0x080L

/*
    Standard MIXes.
*/
#define  MIX_NOC      0x00     /* No change          */
#define  MIX_OR       0x01     /* Src OR Dst         */
#define  MIX_OVER     0x02     /* Src (overpaint)    */
#define  MIX_XOR      0x04     /* Src XOR Dst        */
#define  MIX_LEAVE    0x05     /* Dst (leave alone)  */
#define  MIX_ADD      0x08     /* Src + Dst          */
#define  MIX_SUB      0x09     /* Dst - Src          */
#define  MIX_MEAN     0x0B     /* (Src + Dst) / 2    */

/*
    Flags for CharSetDef structure.
*/
#define MULTIPLANE  0x4000       /* single or multi plane                */
#define PROPSPACE   0x2000       /* single or multi plane                */

/*- - ---- - -- ----- ------------ - -- --------- ---------- ----------

    AI Structures and Typedefs

------------------------------------------------------------------------*/
```

```
typedef  coord_pr     APOINT;                /* Absolute point (in IBMAFI.H)  */
typedef  rcoord_pr    RPOINT;                /* Relative point (in IBMAFI.H)  */


/*
   Font file header
*/
typedef struct CharSetDef                   /* Character Set Definition block */
   {
   UBYTE     reserve1;
   UBYTE     type;
   DWORD     reserve2;
   UBYTE     reserve3;
   UBYTE     cellwidth;
   UBYTE     cellheight;
   UBYTE     reserve4;
   WORD      cellnbytes;
   WORD      flags;
   WORD FAR *indextbl;
   UBYTE FAR *enveltbl;
   UBYTE     cdptlst;
   UBYTE     cdptlast;
   UBYTE FAR *chardef1;
   WORD      reserve5;
   UBYTE FAR *chardef2;
   WORD      reserve6;
   UBYTE FAR *chardef3;
   }  CHARSETDEF;


typedef struct FontFileDefn
   {
   WORD  no_pages; /* no of code pages in the file (size of page_array) */
   WORD  def_page; /* default page index (into page_array)              */
   WORD  alt_page; /* alternate default page index (into page_array)    */
   struct
      {
      BYTE  code_page_id[4];
      WORD  csd_offset;
      } page_array[1];
   } FFDEF;


/* --------  . ..  .. .. ..                .   . ............ .   --------

   AI Parameter Definitions - sorted by AI command number.

   Some commands, such as HEGS, do not have data structures. All commands
   that do are listed here. They can be found by searching for the
   command name followed by "_DATA". Hence, the data structure for
   HINIT is called HINIT_DATA.
```

```
------------------------------------------------------------ ------------*/

/*
    Define structures for HLINE, HCLINE, HRLINE, and HCRLINE. They are
    set up as macros because they require a size parameter.
*/
#define HLINE_DATA(SIZE)    struct\
                               {\
                                UWORD     length;\
                                APOINT    coords[SIZE];\
                               }

#define HCLINE_DATA(SIZE)   struct\
                               {\
                                UWORD     length;\
                                APOINT    coords[SIZE];\
                               }

#define HRLINE_DATA(SIZE)   struct\
                               {\
                                UWORD     length;\
                                APOINT    coord1;\
                                RPOINT    rcoords[SIZE];\
                               }

#define HCRLINE_DATA(SIZE) struct\
                               {\
                                UWORD     length;\
                                RPOINT    rcoords[SIZE];\
                               }


typedef struct
   {
   UWORD    length;
   APOINT   coord1;
   } HSCP_DATA;


typedef struct
   {
   UWORD    length;
   UBYTE    flags;
   } HEAR_DATA;


typedef struct
   {
   UWORD    length;
   DWORD    index;
```

```
    } HSCOL_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     oflags;
    UBYTE     mode;
    UBYTE     iflags;
    } HOPEN_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     foremix;
    UBYTE     backmix;
    } HSMX_DATA;


typedef struct
    {
    UWORD     length;
    DWORD     index;
    } HSBCOL_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     index;
    UBYTE     reserved;
    UBYTE FAR *def;
    } HSLT_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     index;
    } HSLW_DATA;


typedef struct
    {
    UWORD     length;
    UWORD     index;
    } HSGQ_DATA;


typedef struct
```

```
    {
    UWORD     length;
    DWORD     col;
    UBYTE     function;
    } HSCMP_DATA;


typedef struct
    {
    UWORD     length;
    DWORD     eventid;
    } HINT_DATA;


typedef struct
    {
    UWORD     length;
    APOINT    coord1;
    } HSPATTO_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     width;
    UBYTE     height;
    UBYTE     flags;
    UBYTE     reserved;
    UWORD     img_length;
    UBYTE FAR *image;
    UBYTE FAR *colour;
    } HSPATT_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     pal_id;
    UBYTE     res1;
    WORD      first;
    WORD      count;
    UBYTE FAR *address;
    } HSLDPAL_DATA;


typedef struct
    {
    UWORD     length;
    WORD      l;
    WORD      r;
    WORD      b;
```

```
    WORD      t;
    } HSLS_DATA;


typedef struct
    {
    UWORD     length;
    UWORD     format;
    UWORD     width;
    UWORD     height;
    APOINT    coord;
    WORD      l;
    WORD      t;
    WORD      w;
    WORD      h;
    } HBBW_DATA;


typedef struct
    {
    UWORD     length;
    UWORD     format;
    UWORD     width;
    UWORD     height;
    WORD      l;
    WORD      t;
    WORD      w;
    WORD      h;
    } HCBBW_DATA;

typedef struct
    {
    UWORD     length;
    UWORD     format;
    UWORD     width;
    UWORD     height;
    UBYTE     plane;
    UBYTE     reserved;
    APOINT    coord;
    WORD      l;
    WORD      t;
    WORD      w;
    WORD      h;
    } HBBR_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE FAR *address;
    UWORD     len;
```

```
    } HBBCHN_DATA;


typedef struct
    {
    UWORD     length;
    UWORD     format;
    UWORD     width;
    UWORD     height;
    UBYTE     plane;
    UBYTE     reserved;
    APOINT    source;
    APOINT    dest;
    } HBBC_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     acformat;
    UBYTE     rcformat;
    UBYTE     ndimens;
    } HSCOORD_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     acformat;
    UBYTE     rcformat;
    UBYTE     ndimens;
    UBYTE     result;
    } HQCOORD_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     scformat;
    } HSMODE_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     mode;
    UBYTE     clvl[2];
    UBYTE     atype;
    UBYTE     dtype;
    UBYTE     ac_w;
    UBYTE     ac_h;
```

```
    UBYTE     nplanes;
    UBYTE     width;
    UBYTE     height;
    UBYTE     h_pitch;
    UBYTE     v_pitch;
    UBYTE     moco;
    UBYTE     n_i_lvls;
    } HQMODE_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     atype;
    UBYTE     modes[32];
    } HQMODES_DATA;


typedef struct
    {
    UWORD     length;
    WORD      size;
    WORD      stack;
    WORD      palbufsize;
    } HQDPS_DATA;


typedef struct
    {
    UWORD     length;
    APOINT    coord;
    WORD      width;
    WORD      height;
    } HRECT_DATA;


typedef struct
    {
    UWORD     length;
    UDWORD    grupdate;
    UDWORD    alupdate;
    UDWORD    display;
    } HSBP_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     flags;
    } HCLOSE_DATA;
```

```
typedef struct
   {
   UWORD     length;
   UDWORD    table[8];
   } HXLATE_DATA;


typedef struct
   {
   UWORD     length;
   CHARSETDEF FAR *address;
   } HSCS_DATA;


#define HCHST_DATA(LEN)      struct\
                                {\
                                UWORD      length;\
                                APOINT     coord;\
                                BYTE       string[LEN];\
                                }


#define HCCHST_DATA(LEN)     struct\
                                {\
                                UWORD      length;\
                                BYTE       string[LEN];\
                                }


typedef struct
   {
   UWORD     length;
   UBYTE     colm;
   UBYTE     row;
   UBYTE     across;
   UBYTE     down;
   UBYTE FAR *chars;
   UBYTE     width;
   } ABLKMFI_DATA;


typedef struct
   {
   UWORD     length;
   UBYTE     colm;
   UBYTE     row;
   UBYTE     across;
   UBYTE     down;
   UBYTE FAR *chars;
   UBYTE     width;
```

```
    UBYTE     hilite;
    } ABLKCGA_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     colm;
    UBYTE     row;
    UBYTE     across;
    UBYTE     down;
    UBYTE     colour;
    } AERASE_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     sor_colm;
    UBYTE     sor_row;
    UBYTE     across;
    UBYTE     down;
    UBYTE     des_colm;
    UBYTE     des_row;
    } ASCROLL_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     colm;
    UBYTE     row;
    } ACURSOR_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     start;
    UBYTE     stop;
    UBYTE     attr;
    } ASCUR_DATA;


typedef struct
    {
    UWORD     length;
    UBYTE     font;
    UBYTE     res;
    CHARSETDEF FAR *address;
    } ASFONT_DATA;
```

```
typedef struct
    {
    UWORD     length;
    UDWORD    foreTable[16];
    UDWORD    backTable[16];
    } AXLATE_DATA;


typedef struct
    {
    UWORD     length;
    UWORD     segment;
    } HINIT_DATA;


typedef struct
    {
    UWORD     length;
    UWORD     segment;
    } HSYNC_DATA;


#define HMRK_DATA(SIZE)     struct\
                                {\
                                UWORD      length;\
                                APOINT     coords[SIZE];\
                                }


#define HCMRK_DATA(SIZE)    struct\
                                {\
                                UWORD      length;\
                                APOINT     coords[SIZE];\
                                }


typedef struct
    {
    UWORD     length;
    UBYTE     width;
    UBYTE     height;
    UBYTE     flags;
    UBYTE     res;
    WORD      imlen;
    UBYTE FAR *image;
    UBYTE FAR *colour;
    } HSMARK_DATA;
```

```
typedef struct
    {
    UWORD     length;
    APOINT    coord1;
    } HQCP_DATA;


typedef struct
    {
    UWORD     length;
    DWORD     table[16];
    } HQDFPAL_DATA;


typedef struct
    {
    UWORD     length;
    BYTE      paldata[1028];                /* 1028 is worst case, but the   */
    } HSPAL_DATA;                           /* actual size is in HQDPS.      */


typedef struct
    {
    UWORD     length;
    BYTE      paldata[1028];                /* 1028 is worst case, but the   */
    } HRPAL_DATA;                           /* actual size is in HQDPS.      */


typedef struct
    {
    UWORD     length;
    UBYTE     width;
    UBYTE     height;
    } ASCELL_DATA;


/*------------ ------- ------------   ------- ---------- --- ------ -  ---

    End of AFIDATA.H

--------------- ----------------------- -----  -------  -- - -----  - -*/
```

Appendix F

Listing of Hardware Registers

The VESA-approved 8514/A register names, mnemonics, and addresses listed by register address

Port Address (hexadecimal)	Access	Mnemonic	Description
02E8	R	DISP_STAT	Display Status
02E8	W	H_TOTAL	Horizontal Total
02EA	R/W	DAC_MASK	DAC Mask
02EB	W	DAC_R_INDEX	DAC Read Index
02EC	W	DAC_W_INDEX	DAC Write Index
02ED	R/W	DAC_DATA	DAC Data
06E8	W	H_DISP	Horizontal Displayed
0AE8	W	H_SYNC_STRT	Horizontal Sync Start
0EE8	W	H_SYNC_WID	Horizontal Sync Width
12E8	W	V_TOTAL	Vertical Total
16E8	W	V_DISP	Vertical Displayed
1AE8	W	V_SYNC_STRT	Vertical Sync Start
1EE8	W	V_SYNC_WID	Vertical Sync Width
22E8	W	DISP_CNTL	Display Control
42E8	R	SUBSYS_STAT	Subsystem Status
42E8	W	SUBSYS_CNTL	Subsystem Control
46E8	W	ROM_PAGE_SEL	ROM Page Select
4AE8	W	ADVFUNC_CNTL	Advanced Function Control
82E8	R/W	CUR_Y	Current Y Position
86E8	R/W	CUR_X	Current X Position
8AE8	W	DESTY_AXSTP	Destination Y Position/Axial Step Constant
8EE8	W	DESTX_DIASTP	Destination X Position/Diagonal Step Constant
92E8	R/W	ERR_TERM	Error Term
96E8	W	MAJ_AXIS_PCNT	Major Axis Pixel Count

Port Address (hexa-decimal)	Access	Mnemonic	Description
9AE8	R	GP_STAT	Graphics Processor Status
9AE8	W	CMD	Command
9EE8	W	SHORT_STROKE	Short Stroke Vector Transfer
A2E8	W	BKGD_COLOR	Background Color
A6E8	W	FRGD_COLOR	Foreground Color
AAE8	W	WRT_MASK	Write Mask
AEE8	W	RD_MASK	Read Mask
B2E8	W	COLOR_CMP	Color Compare
B6E8	W	BKGD_MIX	Background Mix
BAE8	W	FRGD_MIX	Foreground Mix
BEE8		MULTIFUNC_CNTL	Multifunction Control
Index 0	W	MIN_AXIS_PCNT	Minor Axis Pixel Count
Index 1	W	SCISSORS_T	Top Scissors
Index 2	W	SCISSORS_L	Left Scissors
Index 3	W	SCISSORS_B	Bottom Scissors
Index 4	W	SCISSORS_R	Right Scissors
Index 5	W	MEM_CNTL	Memory Control
Index 8	W	PATTERN_L	Fixed Pattern Low
Index 9	W	PATTERN_H	Fixed Pattern High
Index A	W	PIX_CNTL	Pixel Control
E2E8	R/W	PIX_TRANS	Pixel Data Transfer

Limits of Liability and Disclaimer of Warranty

The Author and Publisher of this book have used their best efforts in preparing the book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness.

The Author and Publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The Author and Publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Index

0AE8 — Horizontal Sync Start Register 209
0EE8 — Horizontal Sync Width Register 210
1AE8 — Vertical Sync Start Register 213
1EE8 — Vertical Sync Width Register 214
02E8 — Display Status Register 202
02E8 — Horizontal Total Register 203
02EA — DAC Mask Register 204
02EB — DAC Read Index Register 205
02EC — DAC Write Index Register 206
02ED — DAC Data Register 207
4AE8 — Advanced Function Control Register 223
06E8 — Horizontal Displayed Register 208
8AE8 — Destination Y Position/Axial Step Constant Register 226
8EE8 —Destination X Position/Diagonal Step Constant Register 227
8PLANE 217
9AE8 — Command Register 232
9AE8 — Graphics Processor Status Register 230
9EE8 — Short Stroke Vector Transfer Register 239
12E8 — Vertical Total Register 211
16BIT 236, 265, 266
16E8 — Vertical Displayed Register 212
22E8 — Display Control Register 215
42E8 — Subsystem Control Register 219
42E8 — Subsystem Status Register 217
46E8 — ROM Page Select Register 222
82B484 321, 322
82C480 320, 322
82E8 — Current Y Position Register 224
86E8 — Current X Position Register 225
92E8 — Error Term Register 228
96E8 — Major Axis Pixel Count Register 229
340X0-based 17
8514/A market growth 20
34010 23, 25, 26
34020 14, 23, 25, 26

A

A2E8 — Background Color Register 241
A6E8 — Foreground Color Register 242
AAE8 — Write Mask Register 243
ABLOCKCGA 78
ABLOCKMFI 80
ACURSOR 82
Adapter Interface 16, 36, 53, 57
Advanced Function Control register 72, 204, 205, 206, 275
AERASE 83
AFI data 343
AFIDATA.H 69, 157
AI 17, 23, 36, 47, 55, 56, 57, 58, 59, 62, 64, 67, 71, 273, 315, 334
AI Commands 331
AI function parameter block 69
AI text modes 60
all-plane BitBLT 296
all-plane copy 178, 182
alphanumeric 87, 185
Alphanumeric color translate 111
Alphanumeric cursor 111
alphanumeric text 84
arcs 56
Area Fills 107, 244, 311
Area pattern 111, 152
Area-pattern origin 111
ASCELL 84, 185
ASCROLL 85
ASCUR 86
ASFONT 87

ATI 316
ATI Technologies 313, 317, 329
axial step constant 286
AXLATE 88
AXSTPSIGN 226

B

B2E8 — Color Compare Register 246
B6E8 — Background Mix Register 247
Background Color 111, 132, 241, 279, 300
Background mix 111, 149, 247, 262, 279, 296
BAE8 — Foreground Mix Register 250
BEE8 Index 0 — Minor Axis Pixel Count Register 254
BEE8 Index 1 — Top Scissors Register 255
BEE8 Index 2 — Left Scissors Register 256
BEE8 Index 3 — Bottom Scissors Register 257
BEE8 Index 4 — Right Scissors Register 258
BEE8 Index 5 — Memory Control Register 259
BEE8 Index 8 — Fixed Pattern Low Register 261
BEE8 Index 9 — Fixed Pattern High Register 262
BEE8 Index A — Pixel Control Register 263
BEE8 — Multifunction Control Register 253
Benchmarking 23
BGA 21, 22, 54
bit planes 133
BitBLT 39, 91, 93, 94, 97, 99, 173, 178, 179, 182, 235, 244, 245, 254, 295, 299, 319
bits per pixel 49
Bottom Scissors 253
bpp 49
BUFSWP 259
Business Graphics Array (BGA) 18, 21
BYTE 73
BYTSEQ 235
BYTSWP 265

C

CALLAFI.ASM 69, 158
CALLAFI.OBJ 157, 158
caveat emptor 313
CGA 14, 185
Character Definitions 333
character 83
character set 87, 139
characters 78, 80, 85
Chips and Technologies 313, 320, 329
circles 56
Clear 108
Clipping rectangle 111, 255, 257, 258, 279
CLKSEL 223
Close 104
COLCMPOP 246, 264
Color bitmapped fonts 334
Color Compare 134, 246
color-compare operation 264, 279
Color Compare register 48, 249, 264
color comparisons 48, 197
Color Graphics Adapter 12
Color index translate table 111
Color palette 279
Command register 46, 197, 234, 265, 283, 289
Compare function 111
Comparison color 111
coordinate system 279
Coordinates 71
Copy Rectangle 235
Current position 111
Current X and Y Position registers 232
Current X Position register 265
Current Y Position 44
cursors 39, 82, 323

D

DAC 13, 42, 193, 207
DAC Data register 205
DAC Read Index register 207
DATARDY 230
DBLSCAN 215
Default font 111
default palette 113, 122
Destination X and Y 235

Destination Y Position/Axial Step Constant 45
Destination Y register 195
DGSTPSIGN 227
diagonal step constant 286
digital-to-analog converter 13, 42
DISABPASSTHRU 223
DISPEN 215
display memory 43
Double buffering 27
double nuggets 209, 210, 260
DRAM 43, 61, 63, 190, 319
DRAW 237
Draw Line for Area Fill 235
drawing performance 24
dual-sync 29
DWORD 73

E

E2E8 — Pixel Data Transfer Register 265
EGA 185
ellipses 56
Enhanced Graphics Adapter 12
Error Term register 273
extended VGA 32

F

FARPTR 73
Fast-Fill Rectangle 234, 311
FCC 28
Fill Rectangle Vertically 234
fill-rectangle 254
filled rectangle 128
Fixed Pattern 265
Fixed Pattern High 253
Fixed Pattern Low 253
Fixed Pattern registers 247, 263
font-loading 185
fonts 49
Foreground color 111, 135, 242, 279
Foreground mix 111, 149, 262, 279, 296
Function Number 331

G

getafi() 159
GETAFI.ASM 188
GPCTRL 219
GPIDLE 218, 220
Graphics Processor Status register 46, 196, 283
Graphics Processor Status register 321
Graphics quality 111, 140

H

hardware compatibles 55
hardware registers 357
HBAR 89
HBBC 91, 179
HBBCHN 93, 182
HBBR 94, 182
HBBW 97, 182
HCBBW 99, 182
HCCHST 101, 185
HCHST 102, 185
HCLINE 169
HCLOSE 70, 72, 104, 164
HCMRK 105, 173
HCRLINE 106, 169, 173
HDILOAD.EXE 68, 159
Headland Technology 313, 316, 322, 329
HEAR 107
HEGS 108
Hercules 22
Hercules Graphics Card 12
HESC 109
HGC 12
HINIT 110, 160, 167
HINT 112
HLDPAL 113, 166
HLINE 70, 115, 169
HMRK 116, 173
HOPEN 70, 72, 117, 159, 164, 185
HORCFG 259
HORTOG 202
HQCOORD 119
HQCP 121
HQDFPAL 122
HQDPS 123, 159
HQMODE 124, 164
HQMODES 127
HRECT 128, 175
HRLINE 129, 169, 173

HRLPC 130, 173
HRPAL 131, 166
HSBCOL 132, 165, 166
HSBP 133, 167
HSCMP 134, 167
HSCOL 135, 165, 166
HSCOORD 136
HSCP 138
HSCS 139, 185
HSGQ 140
HSHS 142, 168
HSLPC 143, 173
HSLT 144, 173
HSLW 146
HSMARK 147, 173
HSMODE 148, 185
HSMX 149, 167
HSPAL 151, 166
HSPATT 152, 175
HSPATTO 154, 175
HSYNC 155
HSYNCPOL 210
HT250 322, 323, 324
HXLATE 156, 185, 334

I

I/O addresses 46, 196
I/O ports 193
IBM 8512 125
IBM 8514 125
IBM AFI.H 339
IBM Image Adapter/A 58
IBM Personal System/2 Display Adapter 8514/A Adapter Interface Application Developer's Guide 67
IBM Personal System/2 Display Adapter 8514/A Technical Reference 67
IBM's ImageAdapter/A 72
IBMAFI.H 69, 157
IGPIDLE 220
IINVALIDIO 220
Image Adapter/A 17, 37
INC_X 233, 237
INC_Y 233, 236
Industry Standard Architecture 195
Initialize 110
Integrated Information Technologies (IIT) 61, 313, 326, 329
INTERLACE 215
Interlaced 56
interlacing 29
interrupt 112
interrupt vector 7FH 68
INVALIDIO 218
IPICKFLAG 220
IRQ2 220
ISA 195
IVBLNKFLG 220

L

LASTPIX 237
Left Scissors 253
Line and area patterns 39
Line Draw 233
Line Drawing 54, 286, 325
line-drawing type 144
line-drawing width 146
line pattern 173
line-pattern count 130, 143
Line-pattern info 111
Line type 111
Line width 111, 173
LINETYPE 233, 236, 237
Long Vectors 289
lookup table (LUT) 41, 42, 52, 59, 193, 195

M

Mach 8 317, 319
Major Axis Pixel Count 235, 289
marker 105, 116
MASM Version 5.1 267
MCGA 334
MDA 50
MEMCFG 216
Memory available and used on 512K and 1-Mbyte boards 26
Memory Control 253
MGA 12
Micro Channel 35, 36, 55, 194
Microsoft C Version 5.1 157, 267
Microsoft Windows 314

Minor Axis Pixel Count 235, 253
MIX 248, 251, 263
Mix registers 241
MIXSEL 263
monitor 28, 30
Monitor Math 30
MONITORID 217, 275
Monochrome bitmapped fonts 334
Monochrome Display Adapter 12
Monochrome Graphics Adapter 12
Multifunction Control register 45, 195
multisynchronous 28

N
new mode 148
No Operation Performed 233
NOT operator 48
nugget 192, 265, 299

O
ODDBNKENAB 216
offset 73
OS/2 314
OS/2 Presentation Manager 36, 59, 65

P
packed pixel 48, 49
palette 39, 42, 113, 131, 151
Pascal 70
pass-through 52, 61, 275
pattern 59, 154, 319, 324
PCDATA 230, 234, 236, 265, 299
PGC 51
PICKFLAG 218, 220, 221
Pixel Control 253
Pixel Control register 244, 247, 262, 266, 285
Pixel Data Transfer 46, 54, 55, 235, 263
Pixel Data Transfer register 197, 230, 247, 250, 300, 323
PLANAR 238, 266
planar 48, 49
Planar architecture 48
Plane mask 111
PLANEMODE 234, 244, 245, 264, 311
polygon fill 56
polygons 89
polyline 103, 106, 115, 129
Presentation Manager 18, 19
Professional Graphics Controller 51
pseudoplanar 60
PWGA 325

Q
Query coordinate types 119
Query current mode 124
Query current position 121
Query task state 123
Queue 321, 324

R
RAM 190
RAMDAC 42, 193, 321
read mask 244
Read Mask register 234, 263, 264, 311
register 45, 64, 198, 199
register standard 53
register-compatible 37
reverse-engineering 37
RGPIDLE 220
Right Scissors 253
RINVALIDIO 221, 242, 266
ROM Page Select register 45
ROMPAGE 222
RPICKFLAG 218, 221
RVBLNKFLG 218

S
segment
SENSE 202
Set coordinate types 136
Set current position 138
short stroke 184
short-stroke vector 39, 46, 50, 51, 61, 291
Short Stroke Vector Fonts 333
Short Stroke Vector Transfer 235
Short Stroke Vector Transfer register 291
Single-plane expansion 178, 182, 296, 299
SIS Microelectronics 313, 327
SSV 39, 50, 61
SSV fonts 333
SSVDRAW 239

Subsystem Control 45
Subsystem Control register 218, 231, 242, 266
Subsystem Status register 46, 220, 275
super-8514/A 26
super-VGA 32
Synchronize 155

T

terminate-and-stay-resident 68
Texas Instruments 329
Texas Instruments Graphics Architecture 21
Text 101, 102, 184, 185
Text and marker info 111
text cursor 82
TI 19
TI 340x0 56
TI 34010 14, 16, 17, 36, 67, 72, 193, 315
TI-based 53, 62
TIGA 21, 22, 23, 54
Top Scissors 253
transparency 48
Trident 61, 313, 326
Trident Microsystems, Inc. 329
TSR 68
Turbo BitBLTs 325

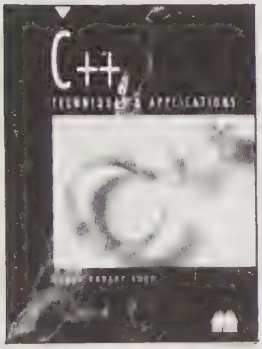
V

VBLANK 202
VBLNKFLG 218, 220
VDADJ 212
VECDIR 239
VESA 16, 32, 40, 54, 198, 199, 315, 316, 317
VESA 8514/A 321
VGA 14, 15, 16, 24, 35, 38, 48, 50, 52, 61, 62, 72, 185, 189, 195, 275, 314, 315, 322, 334
VGA BIOS 314
VGA pass-through 194
Video Electronics Standards 329
Video Electronics Standards Association 32, 315
Video Seven 322
video memory 43
video RAM 43
VRAM 19, 25, 43, 61, 63, 190, 299, 319
VRTCFG 259
VSADJ 213
VSYNCPOL 214
VTADJ 211

WXYZ

Western Digital Imaging 313, 316, 325, 329
Windows 18, 19, 36, 59, 65
WORD 73, 335
Write Mask 243, 249, 279
WRTDATA 230, 238, 244
XOR 48
YMAJAXIS 233, 236

More Programming Tools from M&T Books

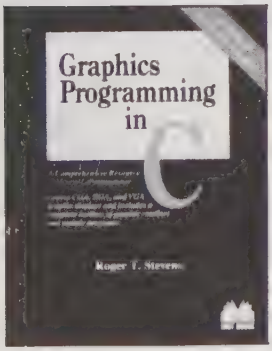


C++ Techniques and Applications

by Scott Robert Ladd

This book guides the professional programmer into the practical use of the C++ programming language—an object-oriented enhancement of the popular C programming language. The book contains three major sections. Part One introduced programmers to the syntax and general usage of C++ features; Part Two covers object-oriented programming goals and techniques; and Part Three focuses on the creation of applications.

Book & Disk (MS-DOS)	***Item #076-1***	***$39.95***
Book only	***Item #075-3***	***$29.95***



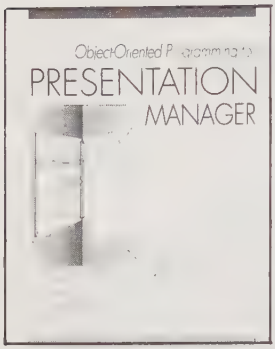
Graphics Programming in C

by Roger T. Stevens

All the information you need to program graphics in C, including source code, is presented. You'll find complete discussions of ROM BIOS, VGA, EGA, and CGA inherent capabilities; methods of desplaying points on a screen; improved, faster algorithms for drawing and filling lines, rectangles, rounded polygons, ovals, circles, and arcs; graphics cursors; and much more! A complete description of how to print hard copies of graphics desplay screens is included. Both Turbo C and Microsoft C are supported.

Book and Disk (MS-DOS)	***Item #019-4***	***$36.95***
Book only	***Item #018-4***	***$26.95***

More C Programming Tools ...



Object-Oriented Programming for Presentation Manager

by William G. Wong

Written for programmers and developers interested in OS/2 Presentation Manager (PM), as well as DOS programmers who are just beginning to explore Object-Oriented Programming and PM. Topics include a thorough overview of Presentation Manager and Object-Oriented Programming, Object-Oriented Programming languages and techniques, developing Presentation Manager applications using C and OOP techniques, and more.

Book & Disk (MS-DOS)	***Item #079-6***	***$39.95***
Book only	***Item #074-5***	***$29.95***

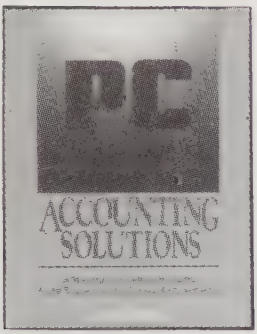
Fractal Programming in C

by Roger T. Stevens

If you are a programmer wanting to learn more about fractals, this book is for you. Learn how to create pictures that have both beauty and an underlying mathematical meaning. Included are over 50 black and white pictures and 32 full color fractals. All source code to reproduce these pictures is provided on disk in MS-DOS format and requres and IBM PC or clone with an EGA or VGA card, a color monitor, and a Turbo C, Quick C, or Microsoft C compiler.

Book & Disk (MS-DOS)	***Item #038-9***	***$39.95***
Book only	***Item #037-0***	***$29.95***

More Programming Tools ...

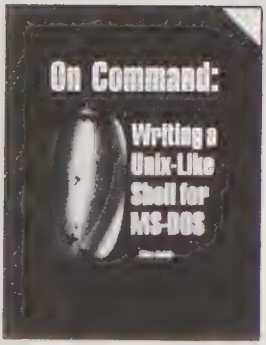


PC Accounting Solutions

by the Editors of *PC Accounting*
(formerly *Business Software*)

PC Accounting Solutions is a well-rounded collection of articles written by accounting experts and is an excellent source of information for managers who want to implement a PC-based accounting system or gain better control of their existing system. Topics include choosing and maximizing PC-based accounting systems, generating and using management accounting reports, developing vertical applications and accounting solutions, forecasting with regression analysis, and more.

Book & Disk (MS-DOS)	***Item #008-7***	***$37.95***
Book only	***Item #009-5***	***$22.29***



On Command: *Writing a UNIX-Like Shell for MS-DOS*

by Allen Holub

On Command and its ready-to-use program demonstrate how to write a UNIX-like shell for MS-DOS, with techniques applicable to most other programming languages as well. The book and disk include a detailed description and working version of the Shell, complete C source code, a thorough discussion of low-level MS-DOS interfacing, and significant examples of C programming at the system level.

Supported features include: read, aliases, history, redirection and pipes, UNIX-like command syntax, MS-DOS compatible prompt support, C-like control-flow statements, and a Shell variable that expands to the contents of a file so that a program can produce text that is used by Shell scripts.

The ready-to-use program and all C source code are included on disk. For IBM PC and direct compatibles.

Book & Disk (MS-DOS)	***Item #29-1***	***$39.95***

Programming the 8514/A

by Jake Richter and Bud Smith

Written for programmers who want to develop software for the 8514/A, this complete reference includes information on both the 8514/A register set and adapter Interface. Topics include an introduction to the8514/A and its architecture, a discussion on programming to the applications interface specification, a complete section on programming the hardware, and more. All sample code and programs are available on the optional disk in MS-DOS format.

Book & Disk (MS-DOS)	***Item #103-2***	***$39.95***
Book only	***Item #086-9***	***$29.95***

To Order: Return this form with your payment to **M&T Books**, 501 Galveston Drive, Redwood City, CA 94063 or **CALL TOLL-FREE 1-800-533-4372** (in California, call 1-800-356-2002). Ask for operator 7085.

❒ **YES!** Please send me the following: ❒ Check enclosed, payable to **M&T Books**.

Item#	Description	Qty	Price

Subtotal _____
CA residents add sales tax __%_____
Add $3.50 per item for shipping and handling _____
TOTAL _____

Charge my ❒ Visa ❒ MC ❒ AmEx
Card No. _____ Exp. Date _____
Signature _____

Name _____
Address _____
City _____
State _____ Zip _____

Note: Prices subject to change without notice. Disks may be returned for exchange only if damaged in transit.

M&T BOOKS

7085

Programming the 8514/A Disk

Save yourself the time and trouble of manual file entry. Contained in this optional disk are 60K worth of code that you won't have to enter yourself.

The **Programming the 8514/A** disk contains the complete set of listings and code, as well as VESA (Video Electronics Standards Association) approved 8514 registered definitions. The disk serves as a good building block for you to develop your own software for the 8514/A standard. Source code is available on 3-1/2 or 5-1/4 disk in MS-DOS format.

To Order: Return this coupon with your payment to **M&T Books**, 501 Galveston Drive, Redwood City, CA 94063 or **CALL TOLL-FREE 1-800-533-4372 (in CA 1-800-356-2002)**. Ask for Operator 7083.

YES! Please send me the Programming the 8514/A source code disk for $20.

CA residents add sales tax ___% _____

Total _____

_____ Check enclosed, payable to M&T Books.

Charge my _____ VISA _____ MC _____ AmEx

Card no. _____ Exp. Date _____

Signature _____

Name _____

Address _____

City _____ State _____ Zip _____

Note: Prices subject to change without notice. Disks may be returned for replacement only if damaged upon receipt.

7083

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT 871 REDWOOD CITY CA

POSTAGE WILL BE PAID BY ADDRESSEE

M&T BOOKS
501 Galveston Drive
Redwood City CA 94063-9925

PLEASE FOLD ALONG LINE AND TAPE CLOSED

GRAPHICS PROGRAMMING FOR THE 8514/A

THE NEW PC GRAPHICS STANDARD

Includes all the detailed information you need to program IBM's 8514/A Graphics Adapter

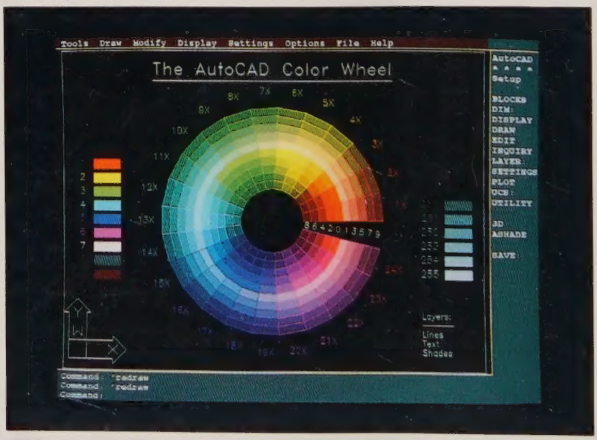
Graphics Programming for the 8514/A unveils the secrets behind IBM's 8514/A Graphics Adapter—a high-resolution graphics standard now being recognized and adopted by leading PC graphics vendors.

Through step-by-step tutorials and sample programs, you'll learn how to program the 8514/A to create impressive PC display graphics. In-depth discussions of the 8514/A introduce you to its architecture and to the details of programming both the Adapter Interface and 8514/A registers. You'll also find complete information on each of the major 8514/A-compatible boards, with a listing of the features and extensions offered by each manufacturer.

Topics include:

- A complete overview of the 8514/A and its compatibles
- Detailed discussions on programming both the AI and the register-level hardware
- An insightful look at the history and future of the 8514/A
- Extensive tutorials and examples

Graphics Programming for the 8514/A includes numerous illustrations, examples, and code samples. Beginners can use the examples as they are, intermediate programmers can adapt them to fit their needs, and advanced programmers can use them as inspiration for new approaches to writing better, faster, more portable drivers. All source code is available on disk in MS/PC-DOS format.



This AutoCAD color wheel, created with Western Digital Imaging's 8514/A-compatible board, illustrates the tremendous range of colors offered by 8514/A technology.

Jake Richter, a noted graphics industry expert, is president of Panacea, Inc., a PC graphics consulting company, and is the graphics columnist for *Personal Workstation* magazine.

Bud Smith is the West Coast technical editor for *Personal Workstation* magazine, and is the author of several books, including *Programming the Intel 80386* (Scott, Foresman and Co., 1987).

This book is for you.
See page 1

M&T BOOKS

52995

9 781558 510869

ISBN 1-55851-086-9

>$29.95